Acceleration of Automatic Speech Recognition for Low-power Devices

Dennis Pinto Rivero

Doctor of Philosophy

Department of Computer Architecture
Universitat Politècnica de Catalunya

Advisors: Jose-Maria Arnau, Antonio González

April, 2022

Barcelona, Spain

Abstract

Machine learning approaches are transforming the landscape of computing and beyond. Among them, *Automatic Speech Recognition (ASR)* is specially eye-catching. Ingrained in the public imagination as the cornerstone of human-machine interaction, speech recognition is set to become a game-changing technology with potential for broad adoption among the general public. Fueled by the remarkable accuracy obtained by current ASR systems, it will likely disrupt, sooner than later, how we interact with all kinds of machines. Today, we are experiencing the early stages of this disruption as can be seen in examples such as smart speakers and AI assistants, dictation engines embedded in virtual keyboards and devices for real-time language translation.

It requires discipline to avoid getting carried away by the promises of a once science-fiction technology. The prospects are, indeed, existing. However, before ASR can momentarily drop the jaws of users in awe and then slowly fade into the background of amazing technologies taken for granted, there are still many challenges ahead. ASR engines are powered by extremely expensive algorithms in terms of computational cost. Decoding a second of speech takes in the order of billions of arithmetic operations. In contrast, devices such as smartphones, smartwatches and other wearables, which are likely the perfect fit for ASR, generally rely on small batteries and operate with very low power. Hence, they often lack the computing power necessary for real-time ASR. Consequently, the current approach to decode speech consists of performing ASR in powerful server computers, relegating the edge device to the role of merely capturing the audio signal. However, given the limitations of this approach, it is seen by many as a workaround until ASR can be completely deployed on the edge.

In this thesis, we study the challenges preventing ASR deployment on edge devices and propose innovations to tackle them, hopefully moving the technology a step forward to the future. First, we characterize state-of-the-art hybrid DNN-HMM. In Hybrid DNN-HMM, the transcription is obtained by searching for the most likely sequence in a large graph that contains every possible transcription. The signal is broken down into frames and then a neural network analyzes each of them. The neural network computes a score for each node of the graph. These scores represent how well each node matches the audio frame. After analyzing the bottlenecks of this ASR system, by characterizing its execution on a representative low-power platform, we propose a heterogeneous platform that contains an accelerator to perform the DNN inference and another accelerator to perform the graph search. This approach results in 4.5x faster execution and 4.3x less energy consumption when compared to a baseline CPU-GPU low-power platform.

To further improve the performance of ASR, we then look into the run-time properties specific to ASR. When ASR is executed on the edge, it is generally expected to decode audio in a streaming way, generating and expanding a partial transcription while the user is still speaking. In this

case, the ASR system analyzes the signal frames one at a time. During each of these decoding steps, it generates a set of partial transcriptions or transcription hypotheses. These hypotheses are expanded during the decoding step by appending new symbols. Additionally, some hypotheses converge into a single hypothesis while others diverge into a multitude of different hypotheses. This creates a difference between decoding steps. During some decoding steps, the system contains many hypotheses whereas, in other steps, the number of hypotheses may be significantly lower. We determine that the number of hypotheses, which can be understood as the confidence of the ASR decoder, can be leveraged by reducing the arithmetic precision of the computations during high confidence steps and increasing it during low confidence steps. According to our results, our technique provides 19.5% reduction in execution time while also reducing the energy consumption by about 16.9%, compared to the heterogeneous platform previously described, with degradation of accuracy below 1%.

Hybrid DNN-HMM ASR systems provide outstanding transcription accuracy even in challenging benchmarks. However, Hybrid DNN-HMM ASR is not the only approach to ASR. End-to-end ASR systems have recently reached state-of-the-art accuracy. The assumptions ingrained in end-to-end systems are more relaxed than those of hybrid systems and the mechanism to train them is more automatic and requires less expert knowledge. For these reasons, many believe that end-to-end systems hold a powerful edge over hybrid systems. For us, that makes them worth studying. These systems are based on large neural networks that are trained to generate the set of hypotheses by themselves. These systems generally include a graph search, as well, but a simpler one. Along with learning how to recognize signal frames, the neural network is trained to learn lexical and language relationships between sounds. Neurons in state-of-the-art neural networks are often activated with a ReLU function, which generates, during run-time, abundant zeros. We design Mixture-of-Rookies, a prediction scheme to detect at run-time when a neuron is going to generate a zero. This technique is embedded in a neural network accelerator that avoids computing the neurons when they are predicted to generate a zero. We estimate that this approach can provide a speedup of 1.21x while consuming 17.7% less energy than a baseline accelerator, for a specific end-to-end system. Furthermore, as many neural networks used for other applications contain ReLU, we evaluate our technique with different neural networks and determine that it can provide significant benefits across the board.

After proposing the previous optimizations, we tackle a different challenge of ASR deployment in low-power devices. ASR is a fast-changing technology and can be expected to continue changing fast as more innovations are proposed. Furthermore, there are many alternative approaches to ASR. Beyond the high-level categories of Hybrid or end-to-end, there are minor variations with often significant impact on transcription accuracy. As more research is conducted, new techniques will keep fine-tuning the algorithms and models for specific use-cases, resulting in a rich collection of alternative systems and techniques for speech recognition. However, this creates an additional challenge for computer architects. An accelerator designed to execute a very specific ASR system will provide huge performance gains but, at the same time, restrict the platform to execute a single ASR system. Furthermore, it risks becoming quickly obsolete, once new ASR techniques hit the market. To tackle this challenge, we propose ASRPU, a programmable ASR accelerator that, taking inspiration from GPUs, enables efficient execution of a wide range of ASR systems and provides a convenient programming model that enables ASR systems to be easily implemented. Our results show that ASRPU can execute state-of-the-art ASR in real-time while consuming less than 1.8 W.

According to our estimations, the entire accelerator fits in about 12 mm^2 when built from 32 nm cell nodes.

The proposals in this thesis represent innovative ways in which computer architecture can push forward the adoption of ASR by the broad public. All these techniques make the execution of ASR systems more amenable for low-power devices, enabling the deployment of state-of-the-art ASR on the edge.

Resumen

Las soluciones basadas en aprendizaje automático están transformando el mundo de la computación. Entre estas, el reconocimiento automático de voz resalta al formar parte del imaginario popular como la forma ideal de interacción con las máquinas. El reconocimiento de voz va de camino a convertirse en una tecnología revolucionaria.

Sin embargo, antes de que los sistemas de reconocimiento de voz puedan utilizarse de manera generalizada, hay muchos desafios que resolver. Los motores de reconocimiento de voz consisten en algoritmos extremadamente caros a nivel computacional. Decodificar un segundo de audio requiere del orden de miles de millones de operaciones aritméticas por segundo. En contraste, los dispositivos que encajan mejor con applicaciones de reconocimiento de voz, como smartphones y weareables, generalmente operan a muy bajo consumo y conectados a una batería, por lo que suelen carecer de la capacidad para ejecutar los sistemas de reconocimiento de voz a tiempo real. Consequentemente, la mayoría de las veces, los algoritmos de reconocimiento de voz se ejecutan en servidores y el dispositivo de bajo consumo tan solo se emplea para capturar el audio y comunicarse con el servidor. Los problemas de seguridad y privacidad, junto con otras limitaciones de este procedimiento, llevan a que mucha gente lo vea como una solución temporal. Idealmente, el reconocimiento de voz sería realizado en el propio dispositivo.

En esta tesis, estudiamos los desafíos que impiden el despliege de los sistemas de reconocimiento de voz en dispositivos de bajo consumo y proponemos innovaciones para solucionarlos. Primero, caracterizamos un sistema estado-del-arte basado en el modelo "hybrid HMM-DNN ASR" ejecutado en una plataforma de bajo consumo. En este tipo de sistemas, la transcripción se obtiene buscando la secuencia con mayor probabilidad en un grafo de decodificación que contiene todas las transcripciones posibles. La señal primero se divide en "frames" y una red neuronal los analiza para obtener una distribución de probabilidad sobre unidades acústicas, llamada "score" acústico. Después de analizar el sistema, proponemos una plataforma heterogénea que contiene una CPU y varios chips específicos para acelerar la búsqueda en el grafo de decodificación y la inferencia de la red neuronal. Al comparar la ejecución en esta plataforma con la ejecución en un sistema base, hemos comprobado que la plataforma heterogénea resulta en una mejora de 4.5x en tiempo de ejecución y una reducción de 4.3x en el consumo de energía.

Para seguir mejorando el rendimiento de esta plataforma, proponemos una técnica que aprovecha una propiedad dinámica de los sistemas de reconocimiento de voz que llamamos "decoder confidence". A medida que el sistema de reconocimiento va decodificando un audio, este va generando transcripciones alternativas o "hipótesis". Para esta técnica, asumimos que cuando el sistema considera un número bajo de hipótesis su "confianza" es alta, y cuando considera un número alto de hipótesis, su "confianza" es baja. Usando esta información, ajustamos la precisión numérica

empleada durante la inferencia de la red neural, ahorrando tiempo de ejecución y energía cuando la precisión es menor, principalmente al reducir los accessos a memoria principal. En nuestros experimentos, esta técnica resulta en una reducción del 19.5% en tiempo de ejecución y una redución del 17.7% en consumo energético, con un aumento del porcentaje de errores en la transcripción inferior al 1%.

En nuestro siguiente trabajo estudiamos un sistema de reconocimiento de voz del tipo ”end-to-end”. Estos sistemas consisten en una red neural más grande y completa que genera directamente las transcripciones. Debido a la simplicidad del entrenamiento de estos sistemas comparado con los sistemas ”hybrid”, junto a otras ventajas que presentan, los sistemas ”end-to-end” han ganado popularidad en los últimos años y muchos de ellos ya alcanzan resultados estado-del-arte. En esta técnica, aprovechamos que la mayoría de estos sistemas emplean funciones ReLU como activación para las capas internas de la red neuronal. Esta función de activación genera un abundante número de ceros en tiempo de ejecución. En esta técnica, proponemos un predictor, llamado ”Mixture-of-rookies” que detecta de antemano cuando la ejecución de una neurona va a resultar en cero. Cuando se da el caso, en vez de ejecutar la neurona, escribimos un cero a su salida, ahorrándonos los cálculos y accesos a memoria que requiere el cálculo de la misma. Para probar que esta técnica se puede aplicar a otros ámbitos, probamos su efectividad en otras redes neuronales de reconocimiento de imágenes, aparte de la red neuronal para reconocimiento de voz. A través de los experimentos que hemos realizado, estimamos que esta técnica proporciona un ”speedup” de 1.21x y una redución del 17.7% en consumo de energía de media.

Tras proponer las optimizaciones anteriores, decidimos enfrentarnos a un desafio diferente. En nuestro último trabajo, diseñamos un accelerador específico para reconocimiento de voz que sea capaz de ejecutar el sistema completo. Nuestro enfoque consiste en proporcionar la máxima flexibilidad, que le permita ejecutar cualquier sistema de reconocimiento de voz, mientras aprovechamos características comunes a la mayoría de sistemas para optimizar su ejecución mediante unidades hardware especializadas. Este acelerador, llamado ASRPU, consiste en un conjunto de unidades de cómputo programables y una serie de unidades específicas y controladores, junto a una jerarquía de memoria optimizada para reconocimiento de voz. De acuerdo con nuestros modelos, una configuración de 1.8W y 12mm^2 puede ejecutar sistemas de reconocimiento de voz estado-del-arte en tiempo real.

Keywords

Automatic Speech recognition, Real-Time, Hardware Accelerator, Low-Power Architecture, Edge computing.

Acknowledgements

I wish to start this dedication by acknowledging and expressing my gratitude to Prof. Antonio González and Dr Jose María Arnau, my advisors, for the role that they played during these years. Thank you for your continuous support and patience and for sharing your extensive knowledge and deep insight. I am very thankful for the opportunity to start my research career at ARCO and for providing me with financial support, for all the fruitful ideas and all the insightful discussions we had.

Again, thank you for the opportunity to work in such an open and creative environment. Your endless enthusiasm and optimism were truly essential to me.

To all the people from the ARCO group. To the professors Jordi, Juan Luis and Llorençs. Thank you for the support throughout these years and the interesting discussions during the group meetings. To Andreas, Franyell and Marc, the post-docs of the group. I am truly thankful for all the interesting conversations we had and the moments we shared. Your paths have been an inspiration for me.

To my wonderful collegues at ARCO: Reza, Martí, Albert, David, Diya, Raúl, Pedro, Jorge, Mohammad, Bahareh, Mojtaba, Rodrigo, Aurora, Imad and Nitesh. I can not emphasize enough how thankful I am for all the good times we shared, for those conversations during lunch, for sharing the toll during the long hours at the lab before deadlines, for the beers (and Fantas) that we had and for those that we could not have, for sharing with me your experiences and your amazing knowledge and insight. I always feel impressed by your achievements and endurance and humbled by how nice and generous your all are. I wish you the best of luck. I am sure that you all will do great at anything you pursue.

To my dear friends, Borja and Germán, and the good times we shared playing online and keeping in touch despite the distance. You always manage to put a smile on my face. Sebas, I want to especially thank you for your support and encouragement during all these years. We have shared so much of our lives and influenced each other so deeply that I can only regard you and Katy as family. I also want to thank my friends Juanlu and Cris for the great moments we shared in Barcelona. It is amazing how life keeps bringing us together again and again. I am sure we will continue to nurture our friendship for many years to come.

To my family. My dear sisters, Sheyla and Yanira, with whom I shared many great memories. Thank you for being there, encouraging me to pursue my dreams. Growing up together with you and learning together how to live and how to be good persons is the best experience I could ask. To my loving mother, Trinidad, who always stood by my side, believing in me. You taught me

the value of being a good person, of accepting and understanding others, no matter what. You are always on the other side of the phone when I most need it. When life feels discouraging and lonely, the memories I have of my childhood with you really spar a beam of light. To my grandparents, Juan and María. You are examples of endurance, patience and good-will. You inspired me to become who I am today. I always looked up to you and I will always continue to do so.

Finally, I admittedly own more than I can give back to my dearest companion, my fiancée, Mengting. For your continuous support, patience and understanding. For your endless love and care. For encouraging me to be brave and confident. You have stood by my side during all these years and for that, I am most grateful and humbled.

The love of my family and friends shaped my identity and my view of the world and for that I could not be more grateful to them. Thank you.

This thesis is dedicated to my mother
for her endless love, support and encouragement.

Contents

1 Introduction **23**

1.1 Motivation 23

1.2 Problem Statement, Objectives and Contributions 25

1.2.1 A Heterogeneous System for low-power ASR 27

1.2.2 Leverage Run-time Decoding Confidence 28

1.2.3 Detect and Remove Ineffectual Computations 29

1.2.4 A Programmable Architecture for ASR 31

1.3 Related Work 32

1.3.1 Early Proposals 32

1.3.2 Beam Search Acceleration 33

1.3.3 DNN Acceleration 34

1.4 Document Organization 35

2 Background on Automatic Speech Recognition **37**

2.1 Automatic Speech Recognition 37

2.1.1 Feature Extraction 38

2.2 Hybrid HMM-DNN 39

2.2.1 Time-Delay Neural Network (TDNN) 41

2.2.2 Decoding Graph 42

2.2.3 Viterbi Decoding 43

2.3 End-to-End ASR 44

2.3.1 Connectionist Temporal Classification (CTC) 45
2.3.2 Time-Depth Separable Neural Network (TDS) 45
2.3.3 Decoding 46
2.4 LM Re-scoring 46

3 Experimental Methodology **49**
3.1 CPU and GPU 49
3.2 Hardware models 50
3.3 Programming frameworks 51
3.4 ASR Benchmarks 52

4 A Low-power Heterogeneous Platform for ASR **53**
4.1 TDNN system 53
4.1.1 Recognition Accuracy 54
4.1.2 Memory 55
4.2 Hardware Platform 55
4.2.1 DNN accelerator 56
4.2.2 Viterbi Accelerator 57
4.3 Experimental Results 59
4.3.1 Execution Time 61
4.3.2 Power Consumption 63

5 Leverage Run-time Beam Search Confidence **65**
5.1 Analysis of Bottlenecks 65
5.1.1 Energy Bottleneck 65
5.1.2 Performance Bottleneck 66
5.1.3 Optimize DNN inference 66
5.2 Dynamic DNN Precision 67
5.2.1 Dynamic Threshold Computation 70

5.2.2 Changes to the DNN Accelerator . . . 71
5.2.3 Changes to the Beam Search Accelerator . . . 73
5.3 Experimental Results . . . 73
5.3.1 Performance Gains of Dynamic Precision AM . . . 73
5.3.2 Effect on Accuracy . . . 75

6 Predicting ReLU outputs to Skip DNN Computations 77
6.1 TDS System . . . 77
6.2 ReLU Activations in DNNs . . . 78
6.3 ReLU Output Predictor . . . 79
6.4 DNN Accelerator with ReLU Output Predictor . . . 86
6.4.1 Control Unit . . . 87
6.4.2 DNN Format . . . 88
6.4.3 Compute Units . . . 89
6.4.4 Binary Prediction Unit . . . 89
6.5 Results . . . 89

7 Programmable Low-Power Architecture for ASR 93
7.1 Architecture of ASRPU . . . 93
7.1.1 Decoding on ASRPU . . . 94
7.1.2 Setup Thread . . . 95
7.1.3 Execution Unit . . . 96
7.1.4 Processing Elements . . . 97
7.1.5 Hypothesis unit . . . 98
7.1.6 Memory Hierarchy . . . 98
7.1.7 Command Decoder . . . 99
7.2 Case Study . . . 99
7.2.1 The Main Process . . . 100

7.2.2 Acoustic Scoring 101
7.2.3 Hypothesis Expansion 101
7.3 Evaluation 102
7.3.1 Methodology and Scope 102
7.3.2 Accelerator Configuration 102
7.3.3 Area and Power 103
7.3.4 Performance 104

8 Conclusions and Future Work 107
8.1 Conclusions 107
8.2 Contributions 109
8.3 Future Work 110

List of Figures

1.1 *Word Error Rate (WER)* of different ASR systems on the librispeech test-clean and test-other benchmarks [7]. 24

1.2 Diagram of a generic Automatic Speech Recognition system. First, a raw signal is transformed into features, and then into acoustic scores. The decoder combines the acoustic scores with the decoding graph to obtain the most likely transcription for the input signal. 26

1.3 Performance of all the utterances in the librispeech test and dev sets decoded by the Kaldi TDNN ASR system on three architectures: CPU, CPU-GPU and CPU-ACCEL. The left-most plot shows the time it took to decode each utterance, where the right-most figure shows the Real Time Factor (RTF). The RTF is the decoding time divided by the utterance time (lower means better). 27

1.4 ASR pipeline on the proposed hardware platform. 28

1.5 Number of hypotheses expanded at each frame during the decoding of 10 seconds of speech. 30

1.6 Speedup obtained by the mixture-of-rookies technique on a set of DNN tasks. 31

1.7 Architecture of ASRPU 32

2.1 Algorithm to compute *Mel-Frequency Cepstral Coefficient (MFCC)*. 39

2.2 Relation between linear frequency domain and Mel frequency domain. 40

2.3 A Time-Delay Neural Network (TDNN) is a chain of Fully connected (FC) layers whose input is a concatenation of several outputs from the previous layer. In this example, each layer depends on the output from the previous layer at t-2, t and t+2. The output of this TDNN at t is computed from the input window (t-8, t+8), which can be seen by extending the dependencies in this diagram. 41

2.4 Example WFST composition between two graphs from Mohri. et al [79]. The button graph is the result of composing the two graphs at the top. Each node in the composed graph is obtained by merging a node from each of the input graphs. For example, the node (1,2) is obtained by merging node 1 from the first graph and node 2 from the second graph. 42

2.5 *Time-Depth Separable Neural Network (TDS)*. 45

2.6 CTC paths for the word *cat* within a sequence of 9 time-steps. The arrows represent example paths. After removing repetitions and blanks (represented by ϵ, all the paths produce the same word) 46

4.1 High level diagram of the complete system. 56

4.2 Architecture of the DNN accelerator based on *DianNao* [20]. In addition to the Neural Function Unit (NFU), it includes three on-chip buffers to store inputs (*NBin*), weights (*SB*) and outputs (*NBout*). The main configuration parameter is Tn, which sets the number of parallel neurons and parallel synapsis per neuron in the NFU. Tn also determines the port width of the memories. 57

4.3 Architecture of the viterbi accelerator. It consists of: several *Issuer* components to load/store data from/to the main memory, each with an associated cache memory; an additional module to compute the likelihood of the paths and two *hash* memories to keep the active tokens for the current and the next frame. 58

4.4 Cumulative distribution of the RTF for all utterances in the test set. The plots correspond to the execution of the kaldi system on the 3 hardware architectures: CPU, CPU-GPU and CPU-ACCEL. 61

4.5 Execution time breakdown for the different ASR components obtained on the 3 hardware architectures. For each architecture, the three bars represent the utterances at percentiles 0, 50 and 100 in the RTF plot (Figure 4.4). 62

4.6 Power dissipation during the computation of the ASR components. Each bar represents the power dissipated by each ASR component, broken down by hardware subsystem. 63

4.7 Cumulative distribution of the energy per frame for decoding for all utterances in the test set. The plot shows the energy for the execution on the 3 hardware platforms: CPU, CPU-GPU and CPU-ACCEL. 63

5.1 Breakdown for energy consumption during ASR evaluation on the mobile SoC presented in Section 4. Chart *(a)* shows the energy breakdown among hardware components during the AM evaluation. Here, it can be seen how reads and writes from the DRAM are responsible for most of the consumed energy, whereas chart *(b)* shows the energy breakdown by ASR component, where the clear bottleneck is the Acoustic Model TDNN evaluation, whereas 66

5.2 Comparison of the WER loss respect to the non-quantized model for various levels of quantization. While quantizing to 8 bits has a small impact on WER, more aggressive quantization sensibly degrades accuracy 67

5.3 WER for a varying percentage of frames evaluated at low precision for a) test_clean and b) test_other. The curves represent the cases when the frames for low-precision evaluation are those with less number of tokens (high confidence), and when they are chosen randomly. 68

5.4 This plot shows the ratio of frames classified for low precision computation in our technique when using a fixed threshold. To obtain this threshold, the train set is evaluated at high precision. Then, all the frames are sorted according to the number of hypotheses expanded during Beam Search. The threshold value is chosen from the number of hypotheses expanded at different percentiles. As we can see, the desired ratio is not achieved in the test sets. 69

5.5 When low precision is used, the number of tokens expanded during Beam Search generally increases. 69

5.6 Threshold value computed by the proposed heuristic compared with the number of tokens expanded by Beam Search. The plot shows a span of 1.2 million frames at $\frac{1}{1000}$ sampling rate. 70

5.7 Schematic of the multiplier unit included in our design. Light grey arrows represent half-precision values. This unit receives two full-precision values, which are interpreted as one full-precision and two half-precision values when operating in half-precision mode. 71

5.8 Schematic of a basic add-tree (5.8(a)) and our duplex add-tree when operating in half-precision mode (5.8(b)). In the latter, each arrow represents a half-precision value. 72

5.9 Frequency of utterances (vertical axis) grouped by the percentage of their frames computed at low precision (horizontal axis). 74

5.10 Energy consumption (a) and execution time (b) normalized to the baseline. The bars in each plot represent: *Worst Utterance*, *Best Utterance* and *Test Set Average* for the baseline and proposed scheme respectively. 75

5.11 Sensitivity analysis of the percentage of frames set as target for low precision evaluation. The curves represent the WER obtained when some percentage of frames is evaluated in low precision for test_clean (a), and test_other (b). 76

6.1 Building blocks for different DNNs. 78

6.2 Percentage of MACs in each type of layer for a set of DNN aplications. 79

6.3 ReLU inputs for binarized neuron (x-axis) versus ReLU inputs for base precision neuron (y-axis). 81

6.4 Distribution of neurons according to the Pearson correlation coefficient of the binary and base-precision ReLU inputs. 82

6.5 Effect of the correlation-based threshold on accuracy loss and percentage of operations saved for different DNNs. 83

6.6 The line perpendicular to A partitions the circle into 2 sectors. Given a random C, the sign of $C \cdot A$ is determined by the partition in which C falls. If another vector B is added (right figure), the circle is partitioned into 4 sectors, which determine the signs of $C \cdot A$ and $C \cdot B$. 84

6.7 Distribution of angles between each neuron and its closest neuron. 85

6.8 Accuracy loss versus percentage of computations avoided for the hybrid *Mixture-of-Rookies* predictor. 86

6.9 Accelerator with support for *Mixture-of-Rookies* ReLU output predictor. 87

6.10 Format to store the DNN in external memory. 89

6.11 Percentage of outputs that are correctly and incorrectly predicted as zero or nonzero by our *Mixture-of-Rookies* predictor. 90

6.12 Performance and energy savings achieved by our *Mixture-of-Rookies* ReLU output predictor compared to the baseline. 90

7.1 Architecture of ASRPU 94

7.2 ASR process executed on ASRPU 94

7.3 Threads in the PE pool 97

7.4 Processing Element (PE) 98

7.5 Size (KB) of each layer of the TDS DNN included in the ASR system. The left plot shows the convolutional layers whereas the right plot shows the fully-connected layers 103

7.6 The left bar plots show the component-level breakdown of area and peak power of ASRPU. The right plots show the distribution of static and dynamic power 104

7.7 Execution time for the TDS ASR system running in ASRPU 104

List of Tables

4.1 WER obtained by different state-of-the-art systems on the Librispeech Corpus . . . 54
4.2 Memory requirements for the models of the ASR system. 55
4.3 Characteristics of the CPU 59
4.4 Characteristics of the GPU 59
4.5 Parameters for the DNN accelerator 59
4.6 Parameters for the Viterbi accelerator 60

7.1 Commands provided by the command decoder 100
7.2 Configuration parameters of the accelerator 103

1 Introduction

This chapter introduces the motivations and objectives behind this thesis. The first section provides our motivation to study automatic speech recognition, outlining the importance of low-power speech recognition for future computing systems. The next section describes the scope of our work and our contributions. The following section provides a discussion of related work, particularly, describing those works aimed at accelerating DNN inference and Beam Search, the main components of speech recognition systems. The final section describes the organization of this document.

1.1 Motivation

Around the 1960s, the first graphic interfaces started appearing (with examples such as *Sketchpad* and Stanford's *On-Line System*). During the 1980s, *Apple* and *Microsoft*, among others, popularized graphical interfaces, promoting the widespread adoption of computers among the regular public. Before that, mostly trained professionals and researchers had access to computers, which were operated via written commands. From that point forward, written commands remained only as specialized interfaces for professionals and highly knowledgeable individuals. Graphic interfaces expanded the use cases of computers and made them easier to use and understand.

Similarly, the current expansion of machine perception technologies will make it straightforward to interact with computers and machines, further expanding their range of use cases. Machine perception refers to those technologies that allow computers to interpret sensory data in a way similar to humans. It includes a broad range of technologies, such as robotic vision, tactile perception and automatic speech recognition.

Among the technologies within the machine perception space, *Automatic Speech Recognition*

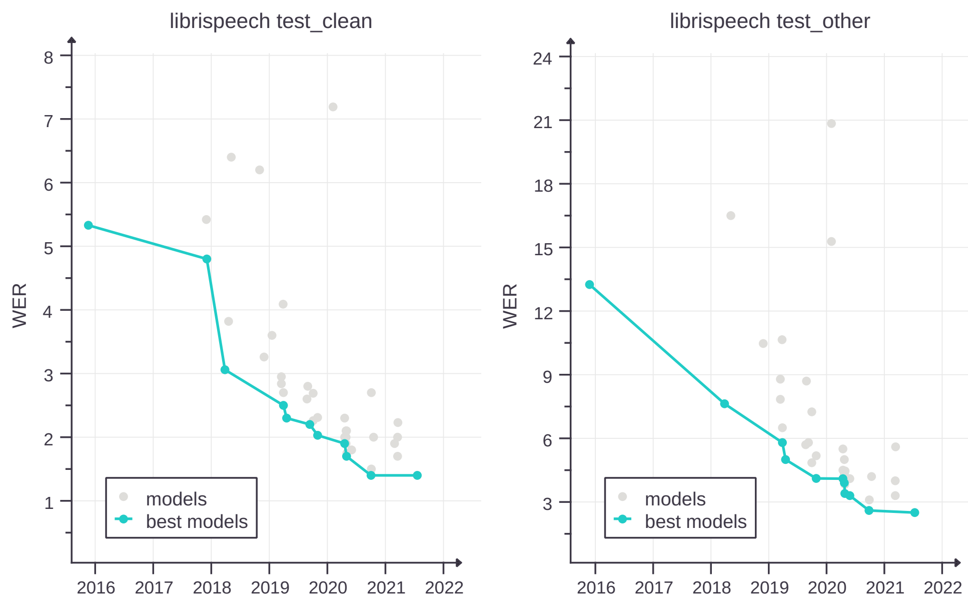


Figure 1.1: Word Error Rate (WER) of different ASR systems on the librispeech test-clean and test-other benchmarks [7].

(ASR) is probably the one with more potential to shape future human-machine interactions. We can currently see the impact of these technologies. Speech-based virtual assistants are very popular devices. Most people are already familiar with Microsoft's *Cortana* [2], Amazon's *Alexa* [1] or Apple's *Siri* [6]. Dictation engines, such as Google's *Text-To-Speech* are built in many smartphone keyboards, online translation tools and search engines [95]. Furthermore, from an economic perspective, the Global speech and voice recognition market is expected to experience very promising growth during the next few years. In a *Fortune Business Insights* report [8] published in 2019, authors forecast a 19.8% *Compound Annualized Grow Rate* in the period from 2019 to 2026.

This unprecedented expansion of ASR is powered by an outstanding increase, during the last few years, in the accuracy of speech decoders. Figure 1.1 [7] shows the *Word Error Rate (WER)* reported in various papers published between 2016 and 2021. DeepSpeech2 [12], published in 2016, obtains a WER of 5.33, i.e. one decoding error per 19 words in the ground truth, whereas in Chang et. al [126], published in 2021, the authors achieve an impressive WER of 1.4, i.e. 1 error per every 71 words, with a *Conformer*-based system.

That level of accuracy is partially responsible for opening ASR for many mainstream uses, as seen by the proliferation, during the last decade, of consumer products based on ASR. In addition to highly accurate, speech recognition must also be fast and reliable. However, given the high compute intensity of ASR, most speech decoding is currently performed on servers [103, 64, 49], which results in unreliable ASR-based services (network connection is not always available) and high latency (specially word-to-word latency). Besides, speech data is personal and must be handled cautiously. Sending speech files to servers raises privacy concerns in a society that is increasingly worried about the handling of personal data by big technology companies.

Many advocate for on-edge ASR as the solution to these problems. However, on-edge decoding does not come without challenges. Edge devices, such as home computers, smartphones and even smart appliances, generally lack the hardware resources and data volumes required to exploit the huge parallelism found in ASR algorithms. Furthermore, streaming decoding, which is a desirable feature for ASR, given the extremely low word-to-word latency it provides, introduces additional challenges by further limiting the amount of parallelism in ASR algorithms. Due to these challenges, it is often unfeasible to deploy highly accurate ASR decoders in edge devices. Nowadays, it is common to deploy smaller and less demanding decoders for on-edge ASR, despite being less accurate.

In this thesis, we aim to improve the performance and efficiency of automatic speech recognition algorithms when constrained to the limited resources of low-power edge devices. We focus on the hardware architecture, generally assuming a heterogeneous architecture with several accelerators in addition to the CPU and a memory hierarchy, and propose innovations to increase performance and reduce energy consumption.

1.2 Problem Statement, Objectives and Contributions

ASR is usually performed in three major steps (Figure 1.2). (1) First, the signal is broken down into overlapping fragments and transformed into a sequence of *feature frames*. (2) Each of these frames is classified into acoustic tokens by an *Acoustic Model (AM)*. For most current systems, the AM is a *Deep Neural Network (DNN)*. The classification of the signal into acoustic units performed by the DNN is not rigid. Instead, the AM classifies the feature frames by generating, for each of them, a vector of probabilities. (3) The last stage, *decoding*, generates a transcription from the acoustic scores. The simplest approach to obtain the transcription takes the tokens with the highest probability on each frame and chains them together. However, more sophisticated approaches, e.g. integrating a lexicon and/or a language model, generally result in better accuracy. When a lexicon or a language model is included, the best scoring transcription can not be obtained by simply taking the single best token for each frame. Instead, a search algorithm, such as *Viterbi Beam Search*, traverses the graph of possible transcriptions (*lattice*) to find the token sequence with the best overall score.

Usually, to maximize efficiency, the input utterance is first captured from beginning to end and then processed. This allows the hardware to maximize data reuse. However, this results in high word-to-word latency because no output is generated until the entire utterance is captured and processed. To achieve low word-to-word latency, the input utterance is processed frame-by-frame (*streaming decoding*), generating partial transcriptions in real-time. Streaming decoding introduces an additional challenge by severely limiting the amount of data reuse available for the hardware. Furthermore, because memory access is the main bottleneck in ASR, the difference in performance between the two options is significant [107].

As previously mentioned, accuracy has kept steadily increasing during the last few years. However, the increase in accuracy is accompanied by an increase in model size, which generally results in higher compute intensity. *Contextnet* [45] is a 31.4M parameter DNN that achieves 2.4 WER on librispeech test_clean, a speech recognition benchmark, and 2.1 WER when scaled up to 112.7M pa-

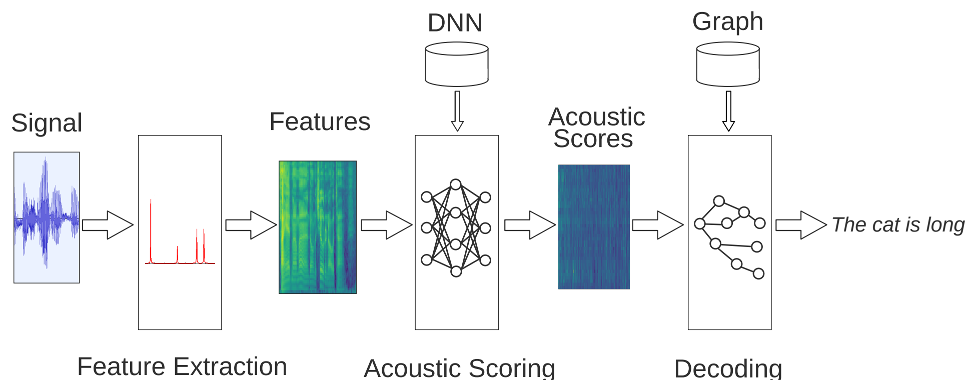


Figure 1.2: Diagram of a generic Automatic Speech Recognition system. First, a raw signal is transformed into features, and then into acoustic scores. The decoder combines the acoustic scores with the decoding graph to obtain the most likely transcription for the input signal.

rameters. Similarly, The *Conformer* [39] network achieves 2.3 and 2.1 WER on the same benchmark with 30.7M and a 118.8M parameters, respectively. Synnaeve et.al [106] train 3 types of networks: a 500M parameter *ResNet* that achieves 2.67 WER, a 500M *Time-Depth Separable (TDS)* network that achieves 2.35 WER and a 296M parameter *Transformer* [108] network that achieves 2.25 WER. Baevski et. al [15] train a 317M parameter transformer network that reach 2.2 WER. *SpeechStew* [19], a 100M parameter DNN, achieves 2 WER, xu et. al [115] achieve 1.7 WER with a 300M parameter network and zhang et. al [126] achieve 1.5 WER with a 1B parameter network.

Another challenge for on-edge ASR comes from the ample diversity of algorithms. Hybrid systems rely on a complex decoding graph formed by composing several independent WFST graphs, including HMMs. On the other hand, E2E systems rely on DNNs and simplify drastically the decoding graph, or even avoid it altogether. The main types of end-to-end systems are those based on *Connectionist Temporal Classification (CTC)* [37] and those based on *seq2seq*[18]. The main difference among them resides in the way they expand hypotheses and compute hypothesis scores.

One of the consequences of these three factors: the fast pace of innovation, high computational cost and vast heterogeneity, is that performing ASR on edge devices is challenging. Consequently, ASR is usually performed on servers, which provide more than enough compute power for the task and where ASR systems can be updated easily. ASR is so computationally expensive that common mobile SoCs are ill-equipped to perform highly accurate decoding within reasonable latency. Hardware acceleration helps in reducing energy cost and latency [120, 107], but, besides high energy-efficiency, the architecture must be flexible enough to accommodate the heterogeneity of current and future systems, while being easily programmable.

In this thesis, we analyze state-of-the-art ASR systems and propose low-power architectural solutions to reduce energy consumption and increase its performance on edge devices. The following paragraphs summarize the different problems we studied and the solutions we proposed to tackle them.

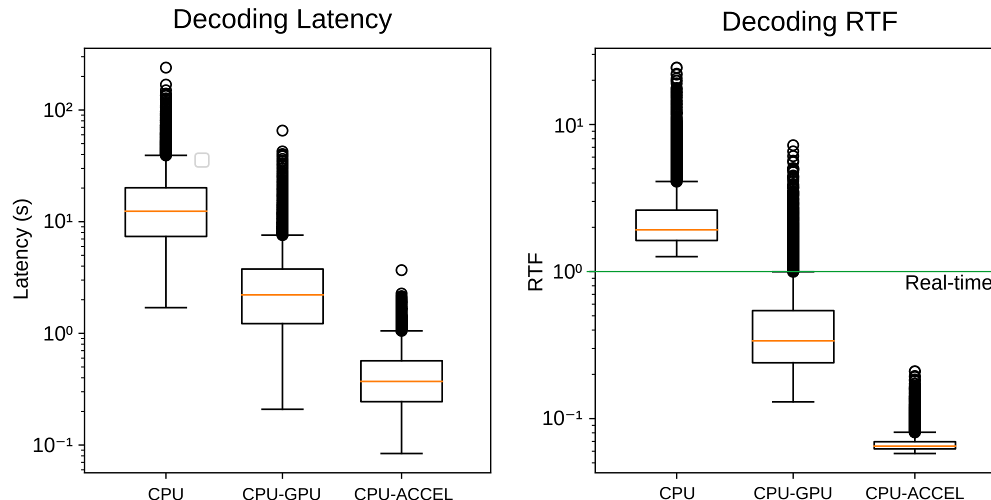


Figure 1.3: Performance of all the utterances in the librispeech test and dev sets decoded by the Kaldi TDNN ASR system on three architectures: CPU, CPU-GPU and CPU-ACCEL. The left-most plot shows the time it took to decode each utterance, where the right-most figure shows the Real Time Factor (RTF). The RTF is the decoding time divided by the utterance time (lower means better).

1.2.1 A Heterogeneous System for low-power ASR

Speech decoding on low-power edge devices is challenging. Figure 1.3 shows the latency for the execution of an ASR system on a Tegra TX1, first on CPU, and then using GPU acceleration. When the ASR is executed on the CPU, all utterances are decoded slower than in real-time, with some of them requiring an order of magnitude longer than real time to decode. When the DNN is accelerated in the mobile GPU, the performance increases significantly, with most utterances decoded in real-time. However, since the GPU only accelerates the DNN, those utterances dominated by the Viterbi Search are not benefited much by the GPU. Consequently, the distribution of latency is broadened. Some utterances are decoded in as little as 0.2 RTF whereas others take up to 7 times real-time.

The decoder is a Hybrid HMM-DNN system implemented in Kaldi [85]. The Acoustic Model DNN in this system is a TDNN network [84]. Decoding is performed over a large graph that combines HMMs, to model tri-phones; a lexicon, to map phonemes to words and a grammar (Language Model) composed of around 200k words. This graph is referred to as *HCLG* graph. In order to alleviate the bottlenecks, we enhance the previous platform with accelerators for the DNN inference and the Beam Search. This heterogeneous SoC contains an ARM CPU, 4 GB of DRAM memory, an accelerator for DNN inference and an accelerator for the Beam Search. The DNN accelerator is based on *DianNao* [20], a popular systolic-array accelerator for DNN inference. The Beam Search accelerator is a scaled-down version of the accelerator proposed on [120].

By offloading the most compute-demanding components of the ASR system to specifically designed accelerators, we overcome the inefficiencies of general-purpose hardware, such as CPUs and GPGPUs. In this case, the CPU is kept idle during the computation of the DNN inference and the Beam Search, saving large amounts of energy and accelerating the decoding process. Similarly, the accelerators are much smaller than the mobile GPU included in the Jetson TX1 board, which results in significant energy savings. Compared to the CPU-GPU platform, the utterances are

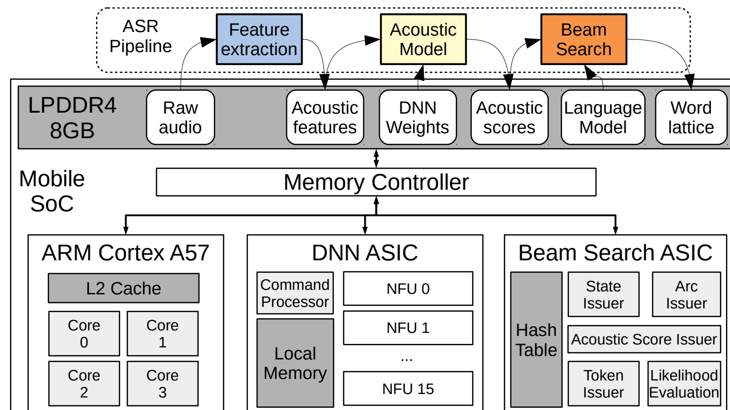


Figure 1.4: ASR pipeline on the proposed hardware platform.

decoded 4.5x faster on the proposed platform, keeping the latency below 0.5 RTF for all the utterances in the test set. Additionally, the energy consumption per second of decoded audio is reduced by 4.3x compared to the CPU-GPU system.

1.2.2 Leverage Run-time Decoding Confidence

Figure 1.4 shows the ASR pipeline executed in our platform. Feature extraction computes the MFCC vectors from raw audio. The acoustic model TDNN performs inference over those MFCC vectors to obtain the acoustic scores, which are combined with the scores obtained from the Language Model (The HCLG graph) to obtain a word lattice. This word lattice can then be re-scored with an additional language model. However, the improvement in accuracy provided by the language model re-scoring step is modest when compared to the significant overhead incurred. For that reason, instead of performing LM re-scoring, we obtain the transcription directly from the lattice.

At this point, the AM DNN inference is the main bottleneck of the system, accounting for, on average, 82% of the execution time and 68.3% of the energy consumption. A popular solution to improve the performance of DNN inference is to decrease the arithmetic precision of the MAC operations. The arithmetic precision used during training is normally 32 bit Floating Point, but it can be reduced down to 8 bit integer. However, reducing it further has a significant effect on accuracy.

In streaming ASR, instead of decoding the entire utterance all at once, or in big batches, the utterance is decoded in small batches of input frames. This leads to higher energy consumption due to lower data reuse, but significantly improves word-to-word latency. During streaming ASR, a small batch of inputs is processed with the AM. Then, the decoder expands the hypotheses, consuming the available acoustic scores. At this point, the best scoring partial hypotheses can be shown in the screen to provide the user with real-time decoding feedback. Additionally, other software components that consume the speech input can start processing the partial transcription in order to decrease response latency.

We make a few observations regarding Streaming ASR: First, the number of hypotheses varies drastically during decoding (Figure 1.5), and second, low accuracy affects different frames differently. Specifically, frames with few hypotheses to expand are less sensitive to low arithmetic precision than frames with higher number of hypotheses to expand. We call the relative number of hypotheses on each frame *run-time decoding confidence*. Frames with many hypotheses have *low confidence* whereas frames with few hypotheses have *high confidence*.

We propose a technique that leverages the previous observation to improve performance and reduce energy consumption. The main idea is to reduce the arithmetic precision for the AM inference when the confidence of the decoder is high. For that purpose, we keep two AM DNNs in memory: The first one is quantized to 8 bits and is used during low-confidence frames. The second DNN is quantized to 4 bits and is reserved for high-confidence frames. To efficiently implement this technique in the hardware, we design dual-precision multiplication and add-tree units, which can operate in both 8-bit and 4-bit precision. They are included in the DNN inference accelerator instead of the regular arithmetic units.

To decide which acoustic model to use for each audio frame, we measure the number of hypotheses generated during the previous decoding step and compare it with a threshold. This threshold is set by measuring the number of hypotheses generated during the decoding of a sub-set of the librispeech train set. We first manually set a threshold that would result in 50% of the frames being scored with the low-precision AM, while the other 50% would be decoded with the high-precision AM. However, we observe that the percentage of frames evaluated at low-precision does not match the expected 50%. This is mainly because changing the precision during acoustic scoring has an impact on decoding, as well. Those frames when the DNN inference for acoustic scoring is performed in low precision, tend to generate less confident scores, which in turn results in more hypotheses being generated (less overall decoding confidence). Because of this, a fixed threshold is not a convenient solution. Instead, we introduce a mechanism to update the threshold in run-time. This mechanism increases or decreases the threshold depending on the average number of frames evaluated in low (and high) precision to keep a fixed ration.

We implemented this technique in the heterogeneous system from the previous section, obtaining further savings in energy and increased performance. Supporting the proposed scheme requires changes in the DNN accelerator, which has to support 8 bit and 4 bit DNN inference and the Viterbi accelerator, which has to register the number of hypotheses expanded during each decoding step and update the threshold.

According to our experimental results, this technique results in 16.9% less energy consumption and a 18.5% reduction in execution time, compared to the baseline platform without run-time adaptation. By restricting low-precision arithmetic operations to high-confidence frames, we can keep the accuracy loss below 1% for test_clean and 1.35% for test_other.

1.2.3 Detect and Remove Ineffectual Computations

Modern DNNs often contain layers activated with *Rectified Linear Unit (ReLU)* functions [12, 48, 39, 66]. The ReLu activation function [62] is like a high-pass filter. When a neuron generates a positive value, it is passed to the next layer unmodified. When it generates a negative value, it

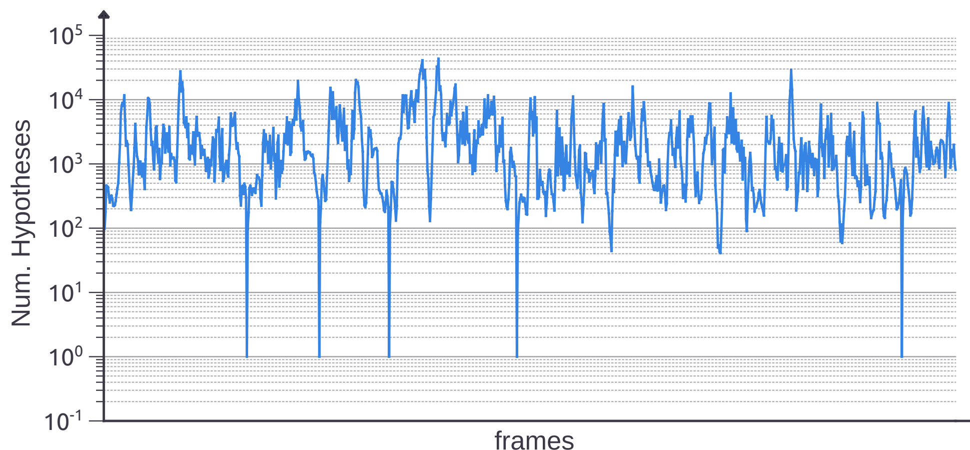


Figure 1.5: Number of hypotheses expanded at each frame during the decoding of 10 seconds of speech.

is clipped to 0. This results in many zeroes being generated.

For example, when decoding with the TDS DNN [84], a DNN for end-to-end ASR, we observed than more than 80% of the outputs of the neurons are clipped to zero. Given that the DNN accounts for the majority of the dissipated energy and execution time, this creates a huge opportunity for optimization.

Several works in the literature propose prediction schemes to detect these zeroes and avoid ineffectual computations. These predictors detect before whether the input of the ReLU will be positive or negative. If the output is predicted to be negative, the neuron is not computed. Instead, a zero is written to memory. There are three main approaches to predict ReLU outputs: Those based on self correlation [71, 102, 17], i.e. correlation between a neuron and the same neuron computed in lower arithmetic precision, those based on spatial correlation [98, 97], i.e. correlation among neurons in the same layer and those based on sub-sampling [10].

In this work, we propose a novel prediction scheme for ReLU activated neurons. Our predictor is composed of two components and the output of a neuron is only predicted to be negative when both components agree on that. The two components are: (1) A spatial-correlation predictor that groups neurons that tend to generate negative values for the same inputs. For every input, it first computes the output of a representative neuron for each group and predicts that the rest of the neurons in the group will have the same outcome. (2) A self-correlation predictor looks at neurons that correlate well with the 1-bit quantized version of themselves. For those neurons, it first computes the output with 1-bit precision. The sign in the resulting value, adjusted to account for shifts in the correlation line, is taken as a prediction for the sign of the value computed with high prediction.

Figure 1.6 shows the results obtained in our experiments with this technique.

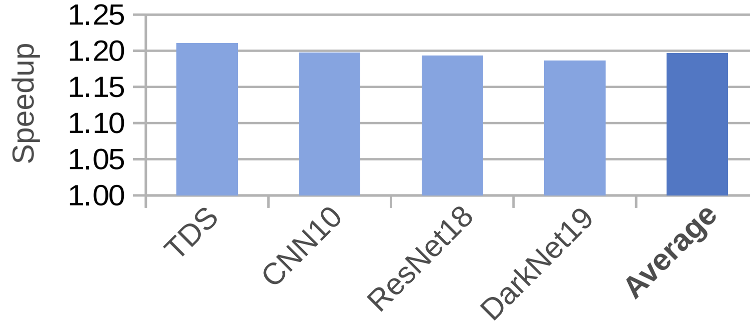


Figure 1.6: Speedup obtained by the mixture-of-rookies technique on a set of DNN tasks.

1.2.4 A Programmable Architecture for ASR

One important challenge for hardware-accelerated ASR comes from the fact that the ASR system space is heterogeneous and it is evolving fast. Consequently, Hardware support for speech recognition must be flexible enough to support many different implementations. Otherwise, it risks becoming obsolete very soon.

However, most proposals in the literature are specific for certain implementations. In this work, we propose ASRPU, a parallel programmable architecture for ASR. This architecture divides the ASR process into two phases: (1) Acoustic Scoring, to transform the raw signal into acoustic score vectors and (2) Hypothesis expansion, to generate new hypotheses from the hypotheses generated during the previous step.

Acoustic Scoring is implemented by the programmer. The programmer writes a set of programs that are launched in sequence by an ASR controller. The programmer can specify how many parallel threads should be launched for each program or it can be determined during run-time via a special *setup* program that is launched right before each program from the sequence. Hypothesis Expansion is implemented by a single program, which is executed several time to expand all the hypotheses generated by the previous decoding step.

Figure 1.7 shows the architecture of the proposed accelerator. It is divided in three major components. The execution units contains the programming elements that execute the kernel threads, the hypotheses unit sorts and prunes the hypotheses generated by the hypothesis expansion threads and the decoding unit decodes commands and configures the accelerator. The ASRPU accelerator contains a d-cache shared among all the PEs. During the execution of the acoustic scoring phase, this memory serves the purpose of a model memory. The programmer can configure the accelerator to prefetch the model data into this memory.

We implemented a state-of-the-art end-to-end ASR system and estimated how long it will take ASRPU to execute it. When configured in a 1.8W setup, the accelerator can execute the ASR system in 2x real-time. This level of performance and the low power consumption makes ASRPU an excellent candidate to include in edge devices.

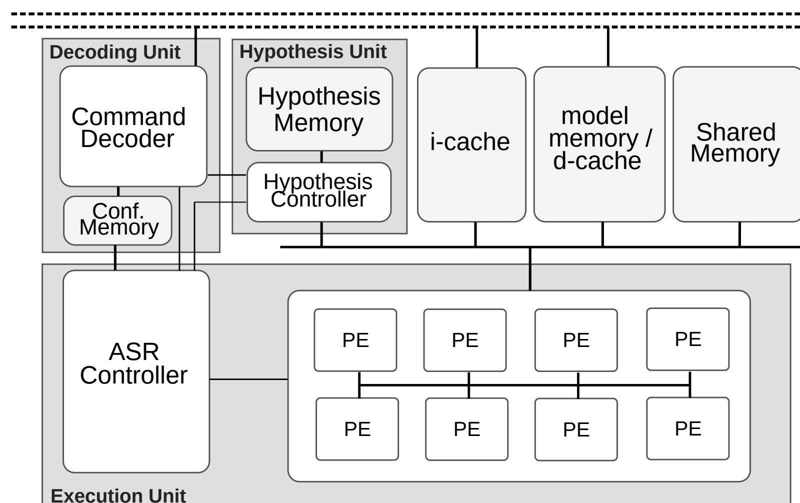


Figure 1.7: Architecture of ASRPU

1.3 Related Work

The literature on hardware acceleration for ASR includes a wide variety of proposals. Some works propose chips that handle the entire ASR process whereas others assume that the ASR process is executed by the CPU and propose accelerators to offload the most compute-demanding components. In this section, we provide a review of related works, focusing on chips for ASR, accelerators for beam search and DNN inference and techniques to improve the performance of those accelerators.

1.3.1 Early Proposals

Early proposals related to hardware acceleration for ASR were mostly focused on HMM-GMM decoders [77, 70, 24]. These systems, generally implemented in *CMU's Sphinx*, contain vocabularies with less than 100k words (e.g. 5k/20k-word Wall Street Journal, 64K-word Broadcast News,...). More recently, Tabani et. al. [107] proposed an accelerator for the *PocketSphinx* system, configured with a 130k word vocabulary and trained with the librispeech benchmark. *PocketSphinx* is a version of *CMU Sphinx* aimed at portability. The accelerator is a $0.94mm^2$, $110mW$ chip and provides a reduction of $5.89x$ in decoding time and $241x$ in energy consumption compared to a mobile GPU baseline. Early HMM-GMM ASR system used to be very popular. However, ASR has gone a long way since then and now most of the ASR systems are either Hybrid HMM-DNN or end-to-end. These systems are primarily limited by the performance requirements of large DNNs and decoding graphs, instead of GMM calculations. In our work, we focus on the challenges of modern ASR architectures. Large DNNs require billions of MAC operations per second and exhibit huge opportunities for parallelization, shifting the performance and energy bottlenecks to the memory access. Also, increasingly large decoding graphs require efficient and adaptable accelerators to handle them. The proposals in this work are aimed at accelerating modern Hybrid and end-to-end

systems. Hence, we propose techniques that do not assume a specific system, but instead leverage untapped properties of ASR systems in general, such as run-time confidence, to alleviate their bottlenecks.

More modern proposals to accelerate HMM-DNN systems include the work by Price et al. [90]. That work proposes a chip that performs from *Voice Activity Detection* (*VAD*) and audio capture, to Viterbi-based decoding. The area of the chip is $13.18mm^2$, and consumes $11.27mW$ (not including power from off-chip components, such as main memory, which is the main bottleneck according to our models) while running a $145k$ word vocabulary benchmark. That work focuses on a single ASR system, which is thoroughly accelerated. However, the current landscape contains plenty of different ASR systems. Furthermore, breakthroughs in the ASR field are not uncommon. Consequently, the main limitation of that work is the rigid design of the chip, that severely restricts its applicability, as it assumes that the DNN fits in the internal memory and only requires a rather limited set of operations. During our work, we abstract away many specific details of the ASR system, designing architectures that support a wide range of systems. For example, in chapters 5 and 6, we propose techniques to leverage run-time decoder confidence and activation sparsity, which can be observed in many ASR systems, and implemented in most accelerator architectures. Finally, in chapter 7, we propose a programmable accelerator that makes very few assumptions about the design of the ASR system.

1.3.2 Beam Search Acceleration

In Yazdani et.al [120], the authors propose an accelerator to offload the Viterbi Beam search performed by Hybrid HMM-DNN systems. Viterbi Beam search can account for more than 80% of the execution time, depending on the ASR system and its configuration. Even when it is executed on a GPU, its contribution on execution time and energy consumption is significant. UNFOLD [117] is another accelerator for Viterbi Beam Search. This accelerator, instead of executing the baseline Viterbi Beam search algorithm, it performs on-the-fly WFST composition to keep the LM separated from the rest of the decoding graph. This provides substantial saving in memory storage and memory accesses. Our work expands from these proposals by exploring the challenges of modern ASR systems. In chapter 4, we include a reduced version of the Viterbi accelerator proposed in [120] in our heterogeneous architecture, along with an accelerator to offload the DNN inference and characterize a modern Hybrid ASR system. Our next work (chapter 5) uses the previous heterogeneous platform and propose changes in both accelerators to leverage run-time decoder confidence.

LAWS [118] is a technique that also leverages run-time decoder confidence. This technique modifies the beam search algorithm to include two beam thresholds and chooses one or the other depending on the decoder confidence. Our approach to leverage decoder confidence is different. Modern ASR systems are not commonly limited by the performance of the Viterbi search, but by the performance of the DNN inference. Consequently, we propose a scheme that uses the decoder confidence to adapt, in run-time, the arithmetic precision used during DNN inference, which results in significant gains in execution time and energy consumption. Our dual arithmetic units provide 2x the throughput when operating at 4 bits, reducing the execution time, whereas accessing the 4-bit DNN weights reduces DRAM bandwidth by half.

1.3.3 DNN Acceleration

Modern ASR systems rely on DNN inference for acoustic scoring. The trend of DNNs is to become increasingly large and complex. Therefore, the contribution of DNN inference in the execution time and energy consumption of ASR systems is also increasing. Consequently, the acoustic model DNN is the major bottleneck in most ASR systems. Commonly, highly parallel computations, such as DNN inferences, are performed in GPGPUs. CUDA [3] and OpenCL [5] are the go-to frameworks for massively parallel execution on GPU and are leveraged by all the major frameworks to develop DNNs, such as *pyTorch* and *TensorFlow*. However, GPUs, given their general-programming focus, tend to waste a significant amount of energy and chip area and fail to leverage properties specific to DNN inference. Therefore, given the current prevalence of DNN inferences, hardware accelerators specifically designed for DNN inference have become increasingly popular.

The most common type of accelerators for DNN inference can be classified as either *Spatial architectures* or *Temporal architectures*. The first category includes *systolic arrays* [31, 55], which are 1D or 2D arrangements of interconnected compute units. At every time step, every compute unit receives data from adjacent units (or the input buffer) and generate results that are passed to other adjacent units or to the output buffer. The compute units for systolic arrays are usually *Multiply-and-Accumulate (MAC)* units, which multiply two operands and add the result to a third operand. The advantage of these accelerators lie in their efficient use of chip area. Compute units are connected to adjacent units, providing large collective bandwidths and potential for parallel execution. However, they are limited in that the design is rather rigid. If the size of the DNN does not match closely the architecture of the systolic array, it incurs in low utilization.

On the other hand, accelerators that implement temporal architectures [20, 73, 21, 4] are more flexible and thus, we think that they provide a better alternative to accelerate ASR systems. This thesis further enhances the performance and energy-efficiency of DNN accelerators by leveraging run-time properties of ASR systems. For example, the scheme in chapter 5 modifies an accelerator based on diannao, a temporal accelerator, exchanging the arithmetic units for dual-precision units that can perform either 4 bits or 8 bits operations. By measuring the confidence of the decoder, we can choose whether to compute the DNN inference in either of the two options.

Other approaches to optimize DNN accelerators leverage either weight sparsity or input/output sparsity widely observed in DNNs [34, 111]. Sparsity in weights means that many weights have a value of '0'. Sparsity in inputs or outputs means that internal results (outputs from internal layers) are commonly '0'. Sparsity allows for more efficient representation in memory which in turn results in more efficient usage of memory bandwidth.

Exploiting weight sparsity is a popular approach to reduce computations and external memory accesses in modern *DNNs*. Many works propose techniques based on generating weight sparsity by zeroing-out less relevant weights [125, 51, 122, 29, 26, 111, 74, 72, 28], which is known as *pruning*. This sparsity is leveraged by using a compact DNN representation that does not include zeros, and including the necessary hardware in temporal accelerators to reconstruct the output.

Related to weight pruning is the idea of exploiting input sparsity [11, 56, 28, 91, 22]. In this approach, the output of the individual layers is generated in a compact sparse format, similar to

the format used for pruned DNNs, which seamlessly allows a *sparse DNN* accelerator [91, 112, 124, 41, 82, 43] to skip more computations.

Other works propose schemes to exploit output sparsity, which is extremely common in ReLU-activated layers [10, 71, 102, 17, 30]. These works generally perform first an approximated computation of the dot-product and, depending on the result, decide whether the output will be negative (and thus, zero after ReLU) or have to be computed in full precision.

Another approach consists on exploiting *Spatial correlation* [98, 97, 60, 32, 75, 61]. However, it has been mainly applied for convolutional neural networks (CNNs), where the neurons within the same layers are correlated since they use the same weights. These works design sophisticated approaches to exploit spatial correlation in CNNs to save computations. However, those techniques rely directly on the kind of spatial correlation observed in CNNs and thus cannot be applied to FC layers, which dominate the computations for many modern DNNs, specially DNNs included within ASR systems. In chapter 6, we propose a technique to leverage output sparsity. We observe that there is also spatial correlation among neurons in FC layers, so we leverage spatial correlation, along with self-correlation, correlation between a neuron and its binary form, to predict when each specific neuron will generate a '0'. The main difference with other approaches is that we combine these two predictor components to build a predictor that works for both CONV and FC layers.

1.4 Document Organization

The remainder of the document is organized as follows

Chapter 2 provides a brief background on Automatic Speech Recognition, introducing all the technical terms and concepts employed through the document.

Chapter 3 describes the experimental methodology employed throughout this work, including the models developed and the tools employed to obtain all the measurements and estimations.

Chapter 4 describes a low-power heterogeneous platform for ASR. This platform is composed of a low-power CPU and 2 accelerators. The first accelerator is for DNN inference and accelerates the DNN inference for acoustic scoring and the RNN for language model re-scoring, as well as some computations from feature extraction.

Chapter 5 describes a technique to modify the bit-precision of the acoustic scoring DNN inference in run-time. In this study, the ASR system decodes utterances in streaming mode, meaning that the input is evaluated in small batches. For every input batch, the hypotheses are slightly expanded. The number of hypotheses at any time step is regarded in this work as the decoding confidence. A dynamically set threshold decides whether input frames are scored using high precision or low precision.

Chapter 6 describes a technique to avoid ineffectual computations from ReLu activated neurons. Our technique consists of a predictor that combines self-correlation and spatial correlation among neurons to determine whether neuron outputs will be '0' or not. Those predicted to be '0' are not computed, saving energy and boosting performance.

Chapter 7 describes the architecture of a programmable low-power unit for ASR. This unit supports a vast variety of ASR systems and provides enough performance for modern ASR systems to be implemented in low-power devices, such as smartphones. We believe that low-power edge devices are the perfect framework for ASR use cases. The proposed architecture is a step forward in the direction of on-edge ASR.

Chapter 8 is a summary of the conclusions extracted during the realization of this thesis. It also includes a discussion on the limitations of this work and promising directions for future research in the area of on-edge ASR and, from a wider perspective, cognitive computing.

2

Background on Automatic Speech Recognition

This chapter provides some background on *automatic speech recognition (ASR)*, including the relevant algorithms and software tools. The first part introduces the overall speech recognition process. The following sections describe the two most prominent approaches for ASR: The hybrid HMM-DNN system and the End-to-End system. Finally, the last few sections provide a brief description of the software frameworks that are common for building speech decoders and some benchmarks frequently used for training and evaluating them.

2.1 Automatic Speech Recognition

Automatic speech recognition (ASR) [92, 121, 109] is the process of decoding an audio speech signal (*utterance*) to obtain a written transcription. Generally, ASR is performed in three steps: *feature extraction*, *acoustic scoring* and *decoding*. The audio signal is received as a sequence of amplitude values. During feature extraction, this input signal is divided into frames, which are then transformed into *feature vectors*. These feature vectors expose the information from the signal frames that is more useful for decoding while getting rid of the rest, making them a better representation of the speech signal than the amplitude sequence. The feature vectors are processed by the *acoustic model (AM)* during the acoustic scoring step. The result of processing the feature frames is a sequence of probability distributions over *acoustic tokens*. The acoustic tokens can be designed as either phonemes, characters or others. The decoding step consists of evaluating different transcription hypotheses to determine which one is more likely to be correct.

The decoding step is performed by traversing a *decoding graph* from the single *start node* to one of the many *end nodes*. The nodes from this graph represent acoustic tokens and the edges, potential transitions between tokens. The edges are weighted in proportion to the probability

of transition. In other words, if the phoneme /'k/ is usually followed by the phoneme /æ/, the edge from the /'k/ node to the /æ/ node will have a high relative score compared to the edges connecting /'k/ to other nodes. The decoding algorithm tries to find the best-scoring path from the start node to one of the end nodes. Several paths are evaluated at the same time. These paths are all expanded one node further for every probability distribution vector (generated by the acoustic scoring step). The score of each path is computed by combining the *acoustic score* that corresponds to the newly appended node, the weight of the traversed edge and the accumulated score from the previous partial path. Once the entire sequence of acoustic vectors is consumed, all the complete paths (those that reached an end node) from the pool of paths are sorted by their score and the best-scoring path is chosen as the transcription.

Currently, there are two main approaches to speech recognition [67, 113]: Hybrid ASR and end-to-end ASR. Hybrid ASR used to be the predominant approach up until recently. However, due to the increasing availability of training data and the increasing sophistication of the DNN models used for AM, end-to-end systems soon reached competitive levels of accuracy. Nowadays, both approaches are present among state-of-the-art systems [7]. However, end-to-end systems are predominant, probably due to their simplicity, particularly during training.

2.1.1 Feature Extraction

As previously discussed, the first step of any speech recognition system is to transform the input signal into a sequence of feature frames. Feature extraction is applied in most speech recognition systems, both hybrid and end-to-end. Different types of features can be employed for speech recognition [59, 99, 27, 40], for example, *Linear Predictive Coding (LPC)* or *Perceptual Based Linear Predictive Analysis (PLP)*. Most of them are supported in the popular frameworks for speech recognition, such as Kaldi [86] and *Wav2Letter++* [87]. However, the most prevalent features are the *Mel-Frequency Cepstral Coefficients (MFCC)*.

Figure 2.1 shows the algorithm to compute MFCC features [44, 54]. The speech signal is first broken down into signal frames. These frames usually consist on 25 ms of audio with an overlap of 15 ms with the previous frame (i.e. a 25 ms sliding frame window with 10 ms shift). In order to improve the spectral quality of the signal, these frames are generally *pre-processed*. The pre-processing usually consists of applying *dithering* and *pre-emphasis* on the signal, among other optional techniques. The next step is to transform the signal frames to the frequency domain via a Fourier transform. However, since the frames start and end abruptly, the power spectrum will contain additional high-frequency components that are not part of the original signal. To remove this effect, the frames are filtered with a *Hamming window* before the application of the Fourier Transform. After the signal is transformed into the frequency domain, the frequencies outside the auditory range are discarded and the rest are squared to obtain the *power spectrum*.

The power spectrum is filtered with a set of triangular filters centred on equally spaced frequencies in the Mel domain to obtain the *mel-frequency filterbanks*. Equally spaced frequencies in the Mel domain correspond to logarithmically spaced frequencies in the linear frequency domain, as shown in Figure 2.2. The rationale behind using the Mel scale is that it matches the sensitivity of the human ear closely than the linear frequency scale. The inner ear receptors differentiate more

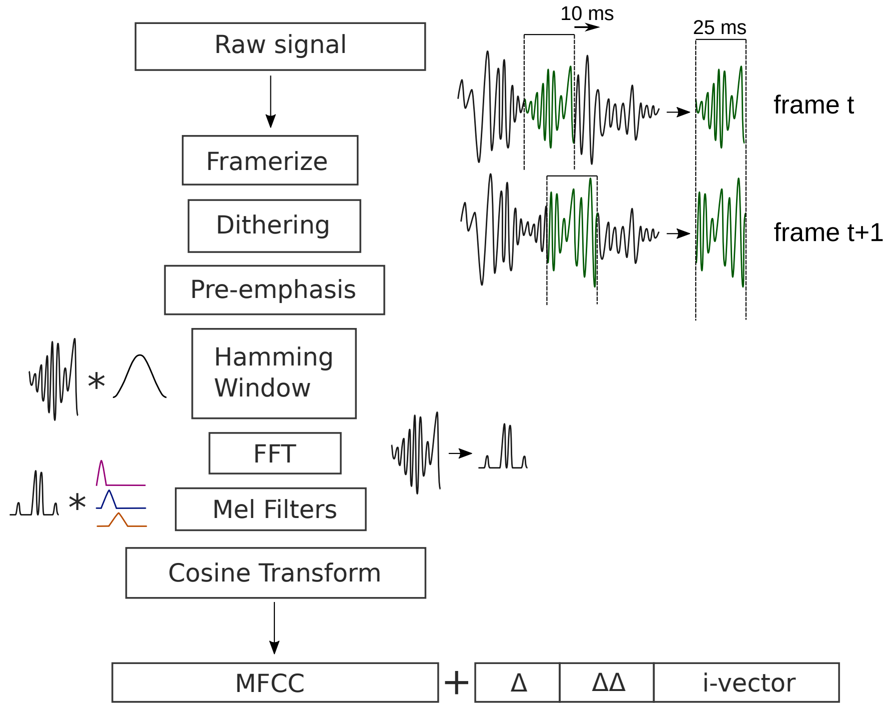


Figure 2.1: Algorithm to compute Mel-Frequency Cepstral Coefficient (MFCC).

levels of frequencies in the lower parts of the audible range than in the higher frequencies [104].

Each Mel filterbank corresponds to one of the triangular filters. Normally, 80 of them are obtained in this step. For each filterbank, the corresponding triangular filter is applied to the frame and the resulting frequency values are summed together. The result is a vector of 80 values, one per filter.

The next step is to transform the filterbanks into *cepstrals* by applying the *Inverse Fourier Transform*, or more commonly, the *Cosine Transform*. Additionally, *liftering* may be applied to the MFCC vector.

2.2 Hybrid HMM-DNN

A Hidden Markov Model (HMM) is a statistical model of a system with non-observable states but observable outputs. The model aims to determine the most likely sequence of states followed by the system given an observed sequence of outputs. The HMM is represented as a graph composed of nodes and edges. The nodes represent the different states of the system, and the edges model the possible transitions between states. The model is defined by its graph topology, which is designed at the beginning and remains unchanged, and the sets of observation probabilities and transition probabilities, which are trained from corpus data.

An HMM can be employed as a generative model of speech [92]. In this case, the HMM models

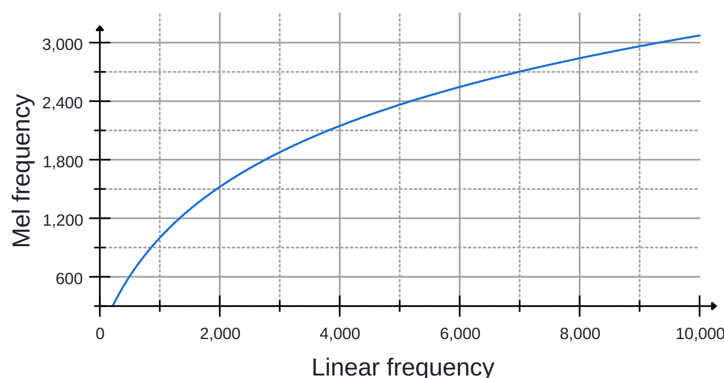


Figure 2.2: Relation between linear frequency domain and Mel frequency domain.

a system capable of generating speech and moving between different states (i.e. configurations). When the system is in each specific state, it is producing the particular sound corresponding to that state, similar to how the larynx and the mouth produce specific sounds depending on their configuration. For example, assuming a system that emits phonemes, for the system to produce the signal for the word "cat" (*/'kæt/*), it has to follow the sequence of states that emit these specific phonemes. Particularly, it has to start in */'k/*, then transition to */æ/* and finally, transition to */t/*. However, the system does not always produce the same signal for the same phonemes. There are many variations that will be recognizable as the same phoneme. The signal variations are modelled by the *acoustic model (AM)*. The AM assigns to each state a probability distribution over signal variations –the probability of observations–.

Given a sequence of observations (sequence of feature frames) and a trained HMM (i.e. the model contains valid probabilities of observations and transitions), the most likely transcription is obtained from the most likely sequence of HMM states. If $\boldsymbol{O} = (\boldsymbol{o_0}, \boldsymbol{o_1}, \boldsymbol{o_2}, ...)$ is a sequence of observations and $\boldsymbol{\pi} = (q_0, q_1, q_2, ...)$ is a sequence of HMM states (i.e. a path through the HMM graph), the likelihood of a path, $\boldsymbol{\pi}$, given an observation, $\boldsymbol{O}$, is the probability, $p(\boldsymbol{O}|\boldsymbol{\pi})$, of making the given observation if the system follows that path. The likelihood of each path is obtained by multiplying the transition and observation probabilities across the complete path.

Prior to the introduction of the hybrid HMM-DNN system, the most prevalent acoustic model was the *Gaussian mixture model (GMM)*, which consists of a mixture of Gaussian distributions for each state. In hybrid HMM-DNN models, [121, 16], instead of GMMs, the acoustic model is implemented through a DNN. By performing inferences over the feature frames with the DNN AM, we can produce a sequence of vectors, each of which represents a probability distribution over all the HMM states (*posterior probabilities*). This DNN has an output neuron for each HMM state. The value at the output of each of these neurons is the posterior probability of the corresponding HMM state given the observation (feature vector).

To train the DNN, the utterances must be labelled with HMM-state-level labels. However, that labelling is specific for each individual HMM system, meaning that no single labelling is universally useful. The usual process to train a DNN-HMM system starts by training a GMM-HMM system. After training, the observation and transition probabilities will contain valid values. By decoding a

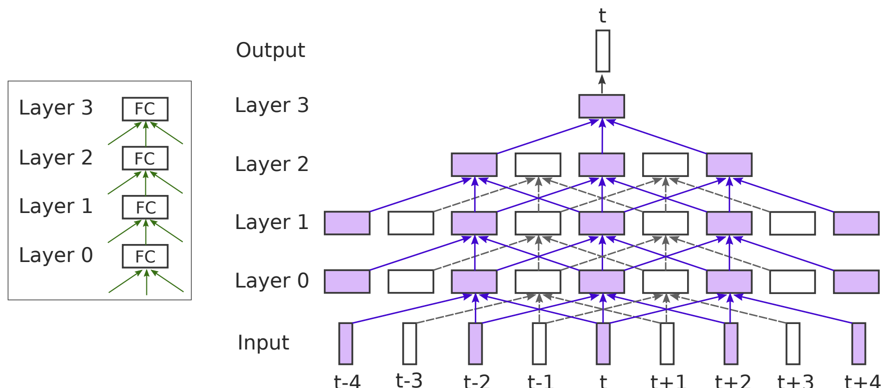


Figure 2.3: A Time-Delay Neural Network (TDNN) is a chain of Fully connected (FC) layers whose input is a concatenation of several outputs from the previous layer. In this example, each layer depends on the output from the previous layer at t-2, t and t+2. The output of this TDNN at t is computed from the input window (t-8, t+8), which can be seen by extending the dependencies in this diagram.

set of utterances with this trained system, we obtain the sequence of HMM states that correspond to the correct transcription for each utterance. These HMM-state-level labels are called *alignments*. The objective now is to train the DNN to classify the feature frames into HMM states according to the alignments obtained from the GMM-HMM system.

Note that by following the previous procedure, the DNN AM learns to generate a probability distribution over the HMM states. However, as previously mentioned, to calculate the likelihood of paths, we need probabilities of observation, which are distributed over signal variations. To interface the DNN and the HMM, the posterior probabilities obtained from the DNN are transformed into likelihoods (in this case, probability distribution over signals) by application of the *Bayes rule*

$$p(\boldsymbol{o_t}|q_t = s) = \frac{p(q_t = s|\boldsymbol{o_t})p(\boldsymbol{o_t})}{p(s)} \tag{2.1}$$

where $p(\boldsymbol{o_t}|q_t = s)$ is the probability of observing $\boldsymbol{o_t}$ while in the HMM state s. $p(q_t = s|\boldsymbol{o_t})$ is the *posterior probability* given by the DNN. In other words, it is the probability of being in the state s while observing $\boldsymbol{o_t}$. $p(s)$ is the *prior* probability of the state s, which is obtained during training, and $p(\boldsymbol{o_t})$ is the prior probability of observing $\boldsymbol{o_t}$. The last term is usually ignored in the computation. That is because it does not depend of the path, and thus, for the same observation, the exact same values of $p(\boldsymbol{o_t})$ are applied to all the paths. Since the likelihood of a path is obtained by multiplying the observation and transition probabilities across the path, all the paths can be divided by the priors of observations without changing the relative likelihood between paths.

2.2.1 Time-Delay Neural Network (TDNN)

Many types of DNNs can be used as the acoustic model for hybrid systems [16, 50, 38, 65]. The straightforward option is to use a fully-connected DNN and leverage its massive potential parallelism for efficient computation. However, fully-connected DNNs usually perform inference over a single input frame but speech frames are not independent, meaning that information from neighbouring frames is useful for decoding. Multiple input frames can be concatenated to provide

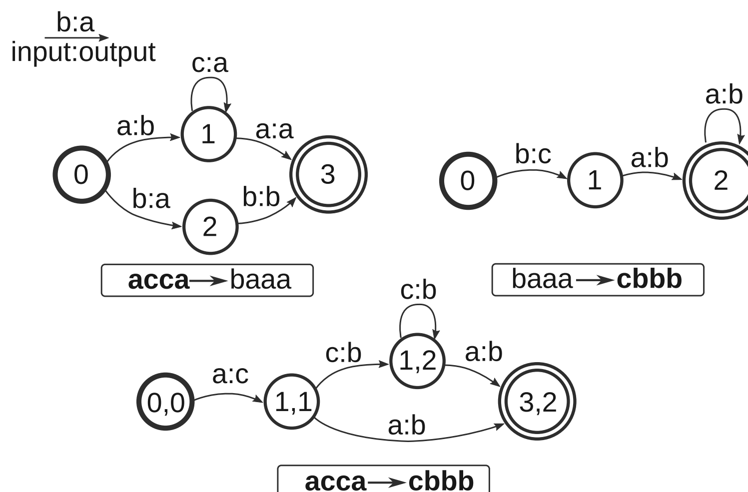


Figure 2.4: Example WFST composition between two graphs from Mohri. et al [79]. The button graph is the result of composing the two graphs at the top. Each node in the composed graph is obtained by merging a node from each of the input graphs. For example, the node (1,2) is obtained by merging node 1 from the first graph and node 2 from the second graph.

a fixed-sized window. However, the number of weights grows very fast with the size of the input in that configuration. *Recurrent Neural Networks (RNN)* provide context information, but their recurrent nature makes them less efficient during inference due to the self-dependencies of the recurrent layers, which severely limits their potential parallelism.

Time-Delay Neural Networks (TDNN) stand out as an effective way to provide long context information to the acoustic model of a hybrid system without the overheads of RNN. The principal advantage of TDNNs over RNNs is that there are no dependencies between timesteps within the same layer of a TDNN network. In other words, layers do not depend on their previous executions. Thanks to that, the amount of parallelism in TDNNs is much higher than in RNNs, making training and inference significantly faster [84], while providing long context information.

Figure 2.3 provides an example of the dependencies between layers on a simple TDNN network. In the example, every layer depends on the output from the previous layer in the time-steps $t-2$, $t-1$, t, $t+1$ and $t+2$. Additionally, dependencies can be sub-sampled, for example, making some layers dependent on only $t-2$, t and $t+2$. This way, the dependency tree is much sparser, meaning that significantly fewer executions of the sub-sampled layers are required in order to perform an inference or training step while keeping the context window equally wide.

The layers in this network are *Fully-Connected* and there are residual connections [48] between some of them.

2.2.2 Decoding Graph

The decoder in *Hybrid HMM-DNN* systems is implemented as a *Weighted Finite-State Transducer (WFST)* [78]. A WFST is a graph that transforms a sequence from a set of labels, the *source*

language to another, the *destination language*. For example, a phoneme-to-word transducer will receive a sequence of phonemes and generate a sequence of words. Each edge in the WFST contains an input label from the source language, and also an output label from the destination language and a weight. The algorithm to transform a sequence from the source language to the destination language is depicted in Figure 2.4. The algorithm traverses the WFST graph starting from a special node, the *start node*. Then it reads the first symbol from the input sequence, traverses the edge with the correspondent input label and generates the symbol from the output label. This process is repeated for the entirety of the input sequence, traversing edges from the graph and generating output symbols until the input sequence is exhausted.

In order to generate sequences of different sizes, both input and output edge labels can be assigned a special symbol, ϵ. An edge with an ϵ input label is traversed without consuming a symbol from the input sequence whereas an edge with an ϵ output label is traversed without generating a symbol for the output sequence. WFSTs commonly contain sub-graphs composed of non-generating edges that either converge or start from an edge that generates the symbol for the entire sub-graph. For example, a phoneme-to-word graph may contain different pronunciations for the same word. Each word is a sub-graph. The first edge of the sub-graph (or the last one, depending on the configuration) produces the word, whereas the other edges consume the different phonemes but they do not generate any output.

The convenience of WFST graphs comes from the functions defined within the WFST framework to operate with them. Particularly interesting for ASR are: *composition*, *determinization*, *weight-pushing* and *minimization*. Composition allows for two graphs to be merged into a single graph. For that, the destination language of the first graph must be the same as the source language from the second graph. Determinization, weight-pushing and minimization are used for compressing the graph.

Generally, on a hybrid system, the HMMs are represented as a WFST graph, which is then composed with other graphs to obtain an *HCLG* decoding graph. This decoding graph is obtained by composition of four graphs: HMM, *context-dependency*, *lexicon* and *grammar*. The HMM graph contains the HMM for the acoustic tokens. The acoustic tokens are usually tri-phones, phonemes with left and right context. The context-dependency graph maps every tri-phone to the corresponding phoneme. The lexicon graph, or pronunciation graph, contains the different pronunciations for each word in the dictionary, mapping from phonemes to words. The grammar, or language model, contains n-grams probabilities, that is, the probabilities of different sequences of words. The sequence sizes range from one (1-gram) to n (n-gram). For example, a 3-gram grammar will contain every 1-gram, 2-gram and 3-gram that can be formed from the vocabulary. The n-gram probabilities are obtained from a text corpus. In order to compress the LM, the less frequent n-grams are generally not included in the language model.

2.2.3 Viterbi Decoding

Once the sequence of acoustic score vectors is computed (by performing an inference pass with the Acoustic Model DNN on the feature frames), the target is to obtain the path through the decoding HCLG graph that most likely transcribes the input signal, given the transition probabilities

(embedded in the decoding graph) and the acoustic probabilities. As previously discussed, the likelihood of a path is obtained by multiplying together the weights of all the traversed links and the acoustic scores corresponding to the traversed nodes from the specific input frame. Additionally, end nodes and word-producing links have additional weights.

The naive algorithm for exploring all the possible paths is extremely expensive. However, the best scoring path can be efficiently found by using the Viterbi search algorithm. *Viterbi search* [16] is a *dynamic programming* algorithm used to obtain the best path through an HMM graph in a single pass. The algorithm has a set of *active nodes*, a set of *next active nodes* and a list of back-pointers. Each iteration, the algorithm consumes the next input from the sequence of acoustic vectors and tries to expand every active node. To expand a node, the algorithm checks all its reachable nodes and puts them in the next active nodes list. Additionally, it creates an entry in the list of back-pointers for each reachable node. These entries refer to the partial paths from the start node to the last explored node. The weight of these paths is computed by multiplying the previous weight in the path by the weight of the traversed link and the corresponding acoustic score from the consumed acoustic vector. The key in the Viterbi algorithm is to merge those paths that converge in the same node at the same time frame. The resulting path will be one of the original paths. Specifically, the path with the lowest overall score (assuming lower score means higher probability).

This process will generally result in a large number of active nodes expanded at many time steps and, consequently, long processing times. A modification of the Viterbi algorithm, the *Viterbi beam search* algorithm, tackles this problem by defining a threshold. At every time step, those paths with a score outside the threshold distance from the score of the best path in the active set are discarded. Additionally, an absolute maximum number of active paths can be fixed. If the maximum is exceeded, the lowest scoring paths are discarded.

2.3 End-to-End ASR

One of the issues of the Hybrid HMM-DNN system is that in order to train the DNN AM, the corpus utterances must be aligned, that is, every frame must be labelled with the corresponding HMM state. Obtaining an HMM-level alignment is not trivial and depends on the HMM itself. As previously discussed, the process to train DNN AMs consists of training first an HMM-GMM model, which does not require alignments. This system is then used to produce alignments for all the utterances in the training set, which are later used to train the DNN.

End-to-End Systems train the DNN from unaligned utterances. That is, the DNN learns the alignments from scratch. These DNNs will learn alignments that minimize a specific cost function and not the alignment that an arbitrary HMM model may expect. These End-to-End systems do not rely on HMMs to decode the system. Instead, the acoustic probabilities are used alone or combined directly with probabilities obtained from pronunciation and language models.

The DNNs for end-to-end ASR are usually *recurrent neural networks (RNN)* implemented with *long short-term memory (LSTM)* or *gated recurrent unit (GRU)* layers. More recent proposals include *attention networks* and *convolutional networks*.

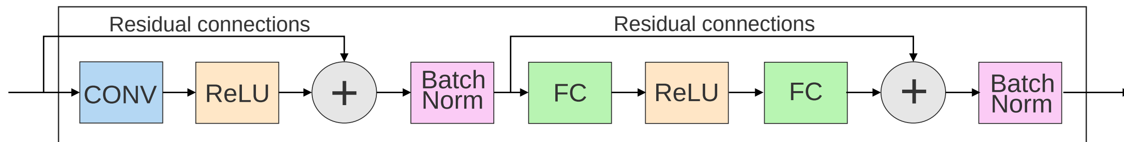


Figure 2.5: Time-Depth Separable Neural Network (TDS).

2.3.1 Connectionist Temporal Classification (CTC)

Connectionist Temporal Classification (CTC) [37] is an objective function used for training DNNs to label unsegmented data sequences. Prior to CTC, unsegmented utterances were labelled by an HMM-GMM system. Labelled utterances (alignments) could then be used to train DNNs for acoustic scoring. CTC DNNs are trained from scratch, with no need for the HMM-GMM system. Note that end-to-end systems do not use HMM, and thus the alignment labels are not HMM states. Instead, the labels are the acoustic tokens directly, which can be phonemes, characters or any other option. For example, *word-pieces (WPM)* [96] have gained popularity during the last few years [87, 23, 45, 106, 46].

In a CTC DNN, the output layer is activated with a softmax function, which guarantees that the output values will comply with the stochastic property (i.e. they all sum to 1), and thus can be used as a probability distribution. The output of the DNN contains a probability value for each label and an additional value to represent the probability of observing 'blank' or no label. During decoding, the DNN processes the input and produces an output for each input feature vector. Since the number of output vectors is equal to the number of input vectors and one of the possible outputs represents the probability of 'no label', the number of labels in the output sequence is equal to or lower than the sequence of input feature vectors. Note that, in addition, by applying sub-sampling in the DNN [84, 46], the number of output vectors can be lower than the number of input feature vectors.

The output sequence is obtained by removing the repeated consecutive labels and blanks. Note that many different output sequences will result in the same label sequence. The likelihood of each label sequence is given by the sum of the likelihoods of all the output sequences that are equal to it. Section 2.3.3 provides more details about CTC paths.

2.3.2 Time-Depth Separable Neural Network (TDS)

Time-Depth Separable (TDS) neural networks are convolutional networks that are used as encoders in *encoder-decoder* architectures (also called *sequence-to-sequence*) [105, 46], or as standalone acoustic models for CTC-based end-to-end systems [89, 106].

The TDS model architecture blocks (Figure 2.5) consist on two sub-blocks, the first one is a 2D convolution over time followed by a *layer normalization* and the second one is fully connected block, composed of two fully-connected layers with a *Rectified Linear Unit (ReLU)* [9] non-linearity in between, and a *layer normalization* [14] at the end. Both blocks contain a *residual connection* [48] between the input of the block and the input of the normalization.

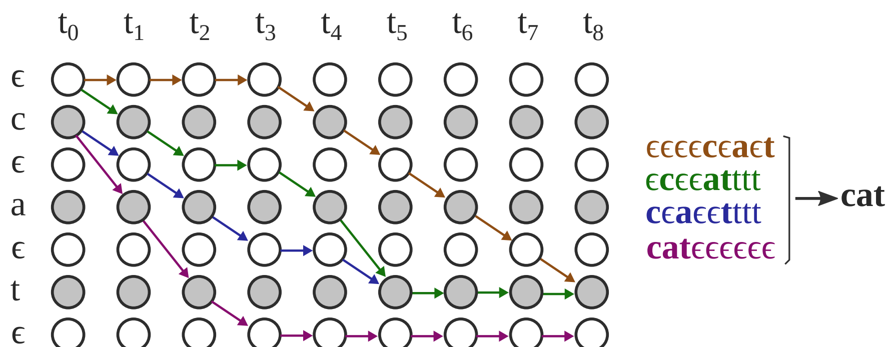


Figure 2.6: CTC paths for the word cat within a sequence of 9 time-steps. The arrows represent example paths. After removing repetitions and blanks (represented by ϵ, all the paths produce the same word)

2.3.3 Decoding

Decoding in an End-to-End system consists of searching for the most likely sequence of labels. Figure 2.6 shows the different sequences that can generate the word *cat*. On CTC, due to removing repeated symbols and blanks in the output sequence, many output sequences generate the same transcription. Because of this, the score of a particular transcription is given by the addition of the score of all the paths that result in that transcription after removing repetitions and blanks. The score of a path is the product of the acoustic and LM probabilities across the entire path. During decoding, paths are compared, and if several paths generate the same transcription, they are merged under the common transcription, summing their scores.

The naive approach for decoding takes the highest-scoring label at each time step and chains them together, removing repetitions and blanks, to obtain the transcription. This approach, however, often results in underwhelming accuracy. In order to improve the accuracy of recognition, additional models, such as pronunciation and language models are usually included in the decoding process. In this case, the sequence of the best individual labels as given by the DNN will not generally be the best scoring sequence. In order to obtain the most likely transcription, many different hypotheses are evaluated in parallel, combining acoustic probabilities with language model probabilities. To avoid the evaluation of an unbearable large number of hypotheses, they are pruned by means of a Beam Search algorithm.

2.4 LM Re-scoring

An n-gram language model gets large pretty quickly when n is increased. In a vocabulary with v words, there are $\sum_i^n v^i$ potential n-grams. Of course, many of them are extremely rare or non-existing within a specific language, meaning that even a conservative pruning pass will yield noticeable results. Additionally, pruning away rare n-grams may result in a significant reduction of its size. However, it comes at the expense of a non-negligible accuracy loss [106].

In any case, composing a large n-gram LM into the decoding graph always results in a significant increases in the size of the graph. To get around that issue, ASR systems usually perform two

decoding passes [93]. The first one is the regular decoding with the decoding graph, but instead of including a large 4-gram LM, for example, they include a heavily pruned 3-gram LM. This first pass produces a set of transcriptions, usually in the form of a graph, called *lattice*. During the second pass the lattice is *rescored* by removing the scores from the first-pass LM and adding the scores from the second-pass LM. This LM can be either a non-pruned 4-gram [106] or a DNN LM [114, 68].

3 Experimental Methodology

This chapter describes the experimental methodology followed throughout this thesis. The first section describes the physical CPUs and GPUs that we use for our experiments and how we obtain measurements of power, performance and energy consumption. The second section describes the models we use to estimate chip area, power, performance and energy consumption of the different accelerators included in our proposals. The third and forth sections describe the software frameworks and the datasets used to build, train and test the ASR systems studied within this work.

3.1 CPU and GPU

Our experiments generally evaluate an ASR system executed on a low-power hardware platform. This platform includes a global DRAM memory, a mobile CPU and one or more accelerators for ASR. The baseline platforms used for comparison include the global DRAM memory, the same CPU and, additionally, a mobile GPU to provide acceleration.

To evaluate a baseline execution, we assume that every component of the ASR system is executed in either the CPU or the GPU. The power, performance and energy consumption are physically measured from the real hardware executing the ASR system. Our experimental platform is a Jetson TX1 board, which contains 4 GB of LPDDR4 DRAM memory, a CPU with 4 ARM Cortex-A57 MPCores and a mobile NVIDIA Maxwell GPU with 256 cores. The board runs on Ubuntu 16.04 LTS with no graphical interface. The CPU contains performance and energy counters to measure different performance metrics and energy consumption. The GPU also contains performance and energy counters. Linux drivers, such as Nvidia-smi, provide access to these registers via linux commands or direct integration in the source code of the ASR system via library

function calls.

Apart from the CPU and the GPU included in our experimental platforms, we performed executions and measurements in several other computers available, usually in a server with 64GB of DDR4 RAM memory, an Intel Core i7-7700K CPU and an Nvidia GeForce GTX1080 GPU with 2560 cores and 8GB of GDDR5X memory.

3.2 Hardware models

In addition to the base platform, all of our proposals include one or more hardware accelerators. The DNN accelerators and the Viterbi accelerators are modelled through cycle accurate simulators. These models simulate the execution of the ASR algorithms from the beginning to the end and provide a cycle count, as well as the activity of the different components, such as the number of reads and writ

In addition to the base platform, all of our proposals include one or more hardware accelerators. The DNN accelerators and the Viterbi accelerators are modelled through cycle accurate simulators. These models simulate the execution of the ASR algorithms from the beginning to the end and provide a cycle count, as well as the activity of the different components, such as the number of reads and writes of a memory component and the number of each arithmetic operation performed on a processing element. Depending on the level of abstraction applied for the implementation of the accelerator, it may also provide functional verification. This is the case of the Viterbi accelerator. For the other accelerators, functional verification is performed separately. For some high level estimations, we use *ScaleSIM* [94] and analytical models.

Performance is estimated by taking the cycle count reported by each simulator and multiplying it by the clock frequency of the simulated accelerator. Some components can be executed in parallel, whereas others follow a serial execution. Data dependencies and synchronizations among different components of the system (accelerators, CPU...) are taken into account to estimate overall execution time.

To estimate chip area and power of the accelerators we rely on several tools. The PEs of the ASRPU are modelled with *McPat* [69] and all the SRAM memories are modelled with the modified version of CACTI [80] included in McPat. These tools reports chip area, access latency, minimum access cycle time, leakage power and the energy cost of both read and write operations. The global DRAM memory is modelled with the *Micron TN5301_LPDDR4 System Power Calculator* [76], using the parameters to model the Z91M package. The model requires the configuration of certain parameters, such as the ratio of write, read and idle cycles and provides static and dynamic power. To model the rest of the components, we implement each of them separately in Verilog and then synthesize them using *Synopsys Design Compiler* with the *Saed32hvt* cell library. Design Compiler provides estimations for chip area and minimum clock period. It also provides power estimations. However, the default assumptions of the tool are generally not very accurate, as stated in the documentation of the tools. Instead of relying on that estimation, we simulate the post-placement circuit with valid random input data and capture the activation factors. Then we use Power compiler to estimate average static and dynamic power during the simulations. Additionally, this

simulation provides a functional verification for the components.

In our experiments, we add together the static power of every hardware component in the platform unless specifically noted. To estimate dynamic energy, we multiply the average dynamic power reported by Power compiler by the cycle time to obtain the average energy consumption when the unit is performing operations. Ultimately, we obtain power and energy estimations by combining the static power of each component, the energy of each operation (memory accesses and arithmetic units), the activations obtained from the cycle-accurate simulators and the execution time.

3.3 Programming frameworks

Kaldi [86] and wav2letter++ [87] are among the most popular software frameworks for building automatic speech recognition systems.

Kaldi is the reference framework to build HMM-GMM and hybrid HMM-DNN systems. It is written in C++ and released under Apache v2.0, a free source licence, which makes extending the code possible for anyone with expert knowledge on ASR and programming. However, extending the code is not necessary for many use cases. Most of the functionality is provided by binaries that are chained together using *bash* statements. ASR systems in Kaldi are built, trained and deployed by writing bash scripts. WFSTs are supported via an external library, *OpenFST*. DNNs are supported via native code built from *BLAS/LAPACK* functions for efficiency and performance. Kaldi is flexible enough to allow for arbitrary HMM and DNN topologies. It also includes several feature extraction implementations, including MFCC, and some techniques for speech pre-processing and speaker adaptation. It can be used to build both streaming and non-streaming ASR systems.

Wav2letter++ [88] is a framework focused on end-to-end ASR systems. It is written entirely in C++ and released under the MIT license. Wav2letter is a component of *Flashlight*, a library for DNN training and inference. Flashlight is built on top of *ArrayFire*, a library for tensor operations. Wav2letter supports arbitrary DNN topologies and both CTC and seq2seq end-to-end systems. The Beam search decoding supports the possibility of including a lexicon and an LM. Graph-based and DNN-based LM are supported. All the Wav2letter systems rely on MFCC features, but the framework provides a wide degree of freedom to modify the parameters of the algorithm.

On this work, we do not propose any new ASR algorithm. Instead, we studied several ASR systems that are proposed in the scientific literature and that are available in any programming framework and performed modifications when needed. For example, on each work we decided whether to perform n-gram LM rescoring, RNN LM rescoring or no rescoring at all. In this thesis we focus on two main ASR systems. The first two works are focused on a Hybrid DNN-HMM system implemented in Kaldi. The scripts to build, train and test this system were already available in kaldi, and most of the techniques included in that system were described in the scientific literature. We used the scripts included in Kaldi to train the model and built our own binaries to test it. We performed other minor modifications to the Kaldi code to perform measurements and quantize the DNN models, among other things. The last two works are focused on an end-to-end model provided in the Wav2letter++ framework. In this case, the trained models were already available

in the website of the wav2letter++ project and we did not have to train them. As in the previous case, we built several binaries and scripts to test the system and perform experiments and slightly modified the wav2letter++ source code to perform measurements and quantize the models.

3.4 ASR Benchmarks

ASR accuracy is frequently measured in *Word Error Rate (WER)* or *Label Error Rate (LER)*. WER measures the word-level edit distance between the transcription given by the decoder and the ground truth, divided by the number of words in the ground truth, and thus giving a relative error ratio. LER is the same idea, but at the label level, instead of the word level. The labels are phonetic units recognized by the acoustic model. In a hybrid system, the labels are HMM states, whereas in a end-to-end system they can be triphones or characters, among others. It is often useful when evaluating end-to-end systems.

There are many speech corpus and benchmarks commonly referenced in the literature. Among the most common ones we have: Librispeech [81], TIMIT [35], Switchboard [36] and WSJ [83]. The ASR systems characterized in this thesis are trained and evaluated in Librispeech. Librispeech is a compilation of 1000 hours of speech, obtained from publicly available audiobooks (from Librivox [58]). The corpus is segmented into 7 sets. Train-clean-100, train-clean-360 and train-other-500 are training sets, dev-clean and dev-other are the evaluation sets used during training, whereas test-clean and test-other are the test set used to measure and report the performance of trained systems.

The different sets are labelled as either “clean” or “other”. To obtain that classification, the authors of librispeech divided the corpus in two, according to the WER obtained during a preliminary decode performed by a system trained on the WSJ corpus. Those utterances with higher WER were labelled as “other” and the rest, as “clean”. The different sets are labelled according to the type of utterances they contain.

Nowadays is common to augment existing train sets by modifying the utterances or by including utterances from other train sets. For example, to train many of the systems in wav2letter++, librispeech is extended with unlabeled utterances from LibriVox, which are included via an unsupervised learning procedure. Language models in wav2letter are trained via Librilight [57], which also contains utterances from both librispeech and Librivox.

4

A Low-power Heterogeneous Platform for ASR

This chapter presents our first work on hardware acceleration for ASR. In this work, we study an HMM-DNN hybrid ASR system and propose a heterogeneous architecture to provide real-time performance on a limited power budget. The platform is based on a market-available SoC, which we enhance with two accelerators. These accelerators undertake the execution of the most compute-demanding components of the ASR system. Our estimations show that the accelerators enable real-time decoding by reducing latency by 4.5x while reducing energy consumption by 4.3x in exchange for a negligible overhead of 3.6 mm^2 in chip area. We start the chapter with an in-deep description of the ASR system. Next, we describe the hardware platform, including the architecture of the accelerators. Finally, the last section consists of a discussion of the proposed platform.

4.1 TDNN system

The objective of this work is to study the challenges in automatic speech recognition when restricted to a very tight power budget, and then propose an heterogeneous platform equipped with the necessary hardware to achieve real-time decoding.

The first step is to choose an ASR system that represents the state-of-the-art in speech recognition. To that aim, we chose one of the hybrid DNN-HMM systems available in the Kaldi framework. This system contains a TDNN network as acoustic model and performs continuous speech recognition with a large vocabulary of over 200k words. It provides very competitive recognition accuracy in Librispeech, both in test_clean and in test_other and is not as demanding, from a computation stand-point, as other comparable systems. Throughout this work, we refer to this system a the *TDNN ASR system* or *TDNN system*.

As already mentioned, the ASR system that we chose is a hybrid HMM-DNN system. Ac-

Table 4.1: WER obtained by different state-of-the-art systems on the Librispeech Corpus

system	E2E	LM	test clean	test other
Human	-	-	5.83	12.69
DeepSpeech2 [12]	Yes	5-gram	5.15	12.73
wav2letter++ [123]	Yes	Conv	3.44	11.24
Jasper DR 10x5 [66]	Yes	Transformer-XL	**2.95**	8.79
pFSMN-Chain [116]	No	TDNN-LSTM	2.97	**7.5**
TDNN	No	TDNN-LSTM	3.67	9.76

cordingly, the transcription hypotheses are obtained by traversing an HMM graph and the acoustic scores are obtained by performing a DNN inference. As described in chapter 2, the decoding process with a hybrid ASR system starts by computing a sequence of feature frames from the utterance. In this case, each feature frame consists of 40 MFCCs obtained from a 25 ms window of the signal shifted by 10 ms for each frame. MFCC frames are complemented with 100-dim i-vectors to provide speaker adaptation. The acoustic model receives the feature frames and computes acoustic score frames. This acoustic scores represent likelihoods over the HMM states. The acoustic model in this system is a TDNN network composed of 24 fully-connected layers, 12 of which are activated by a ReLU activation function followed by a batch normalization layer, whereas the rest do not use any activation function. The TDNN AM contains a total of 18M parameters and requires 773M MACs to decode a second of audio. The input of the TDNN AM consists of 3 feature frames (40 MFCCs each) appended to a 100-dim i-vector and it generates a 6056-dim acoustic score vector.

Decoding is performed by running a Viterbi beam search over an HCLG graph that combines: 3-state HMMs, a 200k word lexicon and a pruned 3-gram language model. The HMMs in this system model triphones. During decoding, the Viterbi search produces a hypotheses lattice, which is re-scored by using a DNN language model before extracting the final transcription. The DNN language model is composed of 3 TDNN and 2 LSTM interleaved layers and a FC output layer. The input and output of the TDNN-LSTM LM are 1024-dim word embeddings. Lattice rescoring is performed as explained in section 2.4. Both the AM and the LM DNNs are quantized to 8 bits.

4.1.1 Recognition Accuracy

The TDNN system is trained and tested in Librispeech. Table 4.1 shows its WER compared to other hybrid and e2e ASR systems. It also includes the accuracy of human transcribers [12] as an additional reference point. The pFSMN-Chain system included in the table is also implemented in Kaldi. The main difference between pFSM-Chain and TDNN is that pFSMN-Chain employs a special CNN network, called *Piramidal CNN* as Acoustic Model, while the TDNN system relies on the TDNN network previously described. We decided to use the TDNN system because TDNN networks are better known and simpler than the pyramidal CNN used in pFSM-Chain. Furthermore, the transcription accuracy of both is very similar, so we think that the TDNN system is a good representation of state-of-the-art hybrid systems.

Table 4.2: Memory requirements for the models of the ASR system.

Component	Memory (MB)	Percentage (%)
Mfcc	0.68	0.07
I–vector	17.95	1.77
Decoding	180.8	17.85
Acoustic Model	16.17	1.6
Language Model	16.12	1.59
Word Embeddings	781.28	77.13
TOTAL	**1013**	**100**

4.1.2 Memory

The TDNN system includes two DNNs, a decoding graph, a few matrices for the MFCC and i-vector computations and a word embedding table that maps each word to its embedding representation.

Table 4.2 shows the size of each component. The size of the two DNNs, along with the matrices for MFCC and i-vector is only 50.9 MB of memory. The decoding graph, after applying aggressive LM pruning and WFST minimization, occupies 180.8 MB, which is very reasonable for a low-power system. The structure that accounts for the majority of the memory footprint is the *Word Embedding Table*, which is the data structure that maps each word to its embedding representation. This table accounts for 77% of the memory footprint is only used during the language model rescoring phase, which is an optional component of the ASR system, and thus can be avoided. However, language model rescoring seems to provide considerable gains in accuracy, so we decided to keep it for this work. Even thought the embeddings table is by far the biggest contributor to memory footprint, it is only accessed sparsely, resulting in a minimum impact in memory traffic.

4.2 Hardware Platform

This section describes the baseline hardware platform that we chose in order to model a low-power system and the accelerators included to achieve real-time ASR. The baseline system consists of a mobile CPU, a DRAM memory, and a mobile GPU. On top of this platform, we include two accelerators. The first accelerator performs DNN inference and the second accelerator performs Viterbi beam search. These accelerators match the ASR algorithms closely, avoiding the overheads of more generic circuits.

Figure 4.1 shows a diagram of the ASR system executed on the platform. Before decoding, all the models are loaded to the DRAM memory, where they are kept throughout the entire process. The DNN accelerator performs the TDNN and LSTM-TDNN inferences, as well as all the vector-matrix and matrix-matrix operation from the feature extraction step, such as the Discrete Fourier

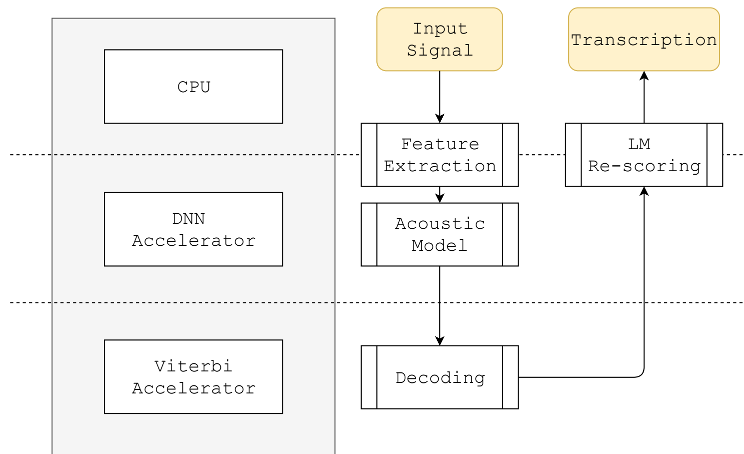


Figure 4.1: High level diagram of the complete system.

Transform and the Cosine transform. The Viterbi accelerator receives the acoustic scores and performs the beam search, traversing the decoding graph, to obtain the transcription lattice. By using these accelerators, the CPU is freed from the most compute-intensive operations, relegating its role to the orchestration of the accelerators and the computation of the operations that are not well suited for them, e.g. some operations from the i–vector computation.

4.2.1 DNN accelerator

The DNN accelerator included in our patform is based on *DianNao* [20], which is a very simplistic and efficient design, very well suited for the kind of DNNs included in the TDNN system. Figure 4.2 is a diagram of the DNN accelerator. It consists of a *Neural Functional Unit* (*NFU*) and several on-chip buffers. The NFU contains all the units required to perform the DNN computations, including an array of adders and multipliers, in addition to special function units to compute the activation functions. The NFU is a pipeline of 3 stages: NFU-1 contains an array of multipliers to multiply each DNN weight with its corresponding input, NFU-2 consists of an add-reduce tree that sums together the resulting values from NFU-1, and NFU-3 contains the special function units to compute the activation functions. Additionally, each NFU stage can be segmented to enable higher clock frequencies. To scale up the design, each NFU pipeline is duplicated Tn times. During run-time, each NFU computes the output of a neuron, allowing for the computations of Tn neurons at the same time.

Regarding the internal memory, it is composed of three SRAM memory buffers: *SB*, to store the DNN weights; *NBin*, to store the inputs, and *NBout*, to store the outputs. These buffers feed data into the NFU. Each SRAM buffer has a DMA that can be used to fetch data in advance, thus hiding the latency of accessing DRAM. Each cycle, the NFU requires Tn inputs and $Tn * Tn$ weights and generates Tn outputs. Consequently, the width for both NBin and NBout is Tn values, whereas the width of SB is $Tn * Tn$ values.

The working principle of the accelerator is as follows: First, inputs are loaded into NBin and

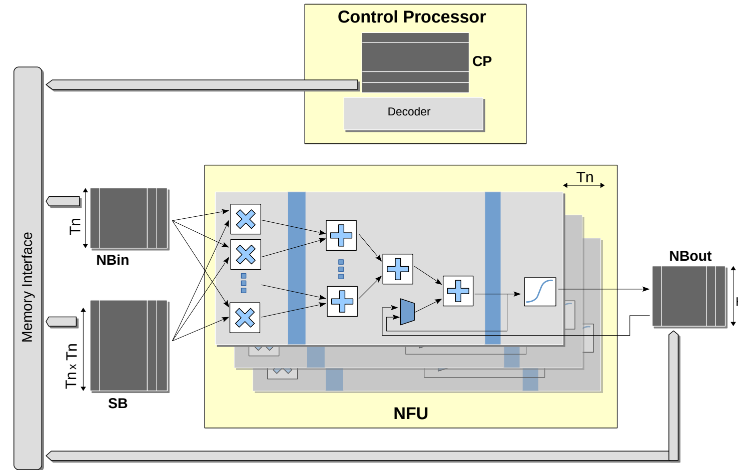


Figure 4.2: Architecture of the DNN accelerator based on DianNao [20]. In addition to the Neural Function Unit (NFU), it includes three on-chip buffers to store inputs (NBin), weights (SB) and outputs (NBout). The main configuration parameter is Tn, which sets the number of parallel neurons and parallel synapsis per neuron in the NFU. Tn also determines the port width of the memories.

DNN weights are loaded into SB. Each cycle, Tn values from the same input vector are broadcasted to all the NFUs. At the same time, Each NFU receives Tn weights from the same neuron. Each NFU computes a different neuron, meaning that SB provides $Tn * Tn$ DNN weights, Tn weights for each of the Tn neurons.

A neuron may have thousands of inputs, whereas in the hardware we have in the order of tens of multipliers. Because of that, NFU-3 is idle while NFU-1 and NFU-2 iterate many times through the input and weight blocks, accumulating the partial result of the neuron. Only when the entire neuron has been computed, NFU-3 computes the activation function for the neuron.

The accelerator contains a *Control Processor* with an additional buffer to store instructions. The control processor fetches instructions from the instruction buffer, decodes them, and generates the control signals for the NFU and the DMAs of the different SRAM buffers. These instructions are the interface between the accelerator and the rest of the SoC. The instruction format contains fields to specify all the parameters required by the matrix-matrix operations available in the accelerator, such as the location of the DNN weights and inputs, their sizes and the activation functions. Moreover, the accelerator can be commanded to reuse the DNN weights or inputs that are already in the accelerator from the previous execution. Thus, reducing memory traffic by maximizing data reuse.

4.2.2 Viterbi Accelerator

The Viterbi search is a graph-processing algorithm that generates sparse and unpredictable memory accesses. On every iteration, it expands the active nodes of the highly irregular decoding graph by traversing their output links. During this operation, only a small fraction of the graph

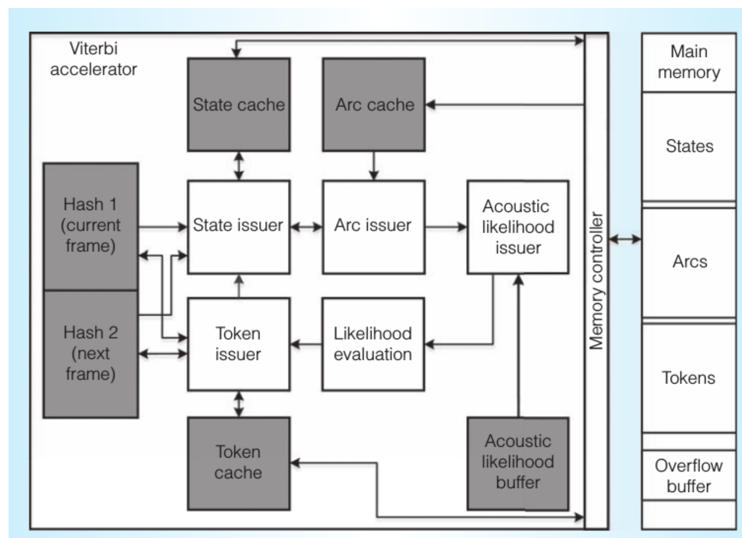


Figure 4.3: Architecture of the viterbi accelerator. It consists of: several Issuer components to load/store data from/to the main memory, each with an associated cache memory; an additional module to compute the likelihood of the paths and two hash memories to keep the active tokens for the current and the next frame.

is accessed. Therefore, the Viterbi Beam Search is not well suited for execution on highly-parallel hardware, such as GPUs or the previous DNN accelerator. Furthermore, it is not a good fit for the CPU either, as caches exhibit poor hit ratios due to the irregular memory accesses. To achieve high-performance and low-power Viterbi search, we included the accelerator described in [120], which is very well suited to execute the Viterbi beam search, and thus provides high performance and energy efficiency.

Figure 4.3 illustrates the architecture of the Viterbi accelerator. It consists of several modules: The *State Issuer* reads an active *token* and fetches the corresponding *node* in the decoding graph from DRAM. The *Arc Issuer* receives the node fetched by the State Issuer and obtains its output links. Nodes and links are independently cached to exploit temporal locality. The *Acoustic Likelihood Issuer* reads the acoustic score associated with the links from the *Acoustic Likelihood Buffer*, which contains all the acoustic scores for the current frame. The next component is the *Likelihood Evaluation*. It computes the cost of traversing each of the output links and generates the new potential hypotheses. The *Token Issuer* receives the new hypotheses and either discards them or stores them in the hash memory to continue expanding the paths in the next frame, depending on whether the costs are within the *beam* or not.

The accelerator contains two hash memories to store the hypotheses for the current and the next frame. These hash memories are swapped, with no memory transfers at the beginning of each frame. If the hypotheses do not fit in the hash memory, they are sent to a reserved space in DRAM, labelled as *Overflow buffer*. Besides that, all the hypotheses generated during each frame are stored in another region of DRAM where they can be accessed to extract the final hypothesis or lattice. The original design from [120] stores independent hypotheses. In order to obtain the lattice, we had to slightly modify the accelerator to store additional data.

Table 4.3: Characteristics of the CPU

Frequency	1.7 GHz
CPU	*ARM Cortex-A57*
Number of Cores	4
Cache	32+48 KB L1, 2 MB shared L2

Table 4.4: Characteristics of the GPU

Frequency	1 GHz
Architecture	*Maxwell*
Number of Cores	256
Cache	48 KB L1, 256 KB L2

Table 4.5: Parameters for the DNN accelerator

Technology	28 nm
Frequency	55 MHz
Bitwidth	1 Byte
Weight Buffer (SB)	64 entries, 16 KB
Input Buffer (NBin)	64 entries, 1 KB
Output Buffer (NBout)	64 entries, 1 KB
Tn	16 values

To deal with the sparse and irregular memory accesses, the Viterbi accelerator includes an area-effective solution based on the Decoupled Access-Execute paradigm [101]. After the pruning step, the addresses of all the arcs that will be accessed during the next iteration are computed in advance and the memory requests are issued early to hide the memory latency.

4.3 Experimental Results

Tables 4.3 and 4.4 show the characteristics of the CPU and the GPU included in the baseline system. The CPU contains on 4 ARM Cortex-A57 cores operating at $1.7GHz$. The GPU is a 256-core NVIDIA Maxwell mobile GPU. The board also contains 4 GB of LPDDR4 DRAM memory. However, instead of the 4 GB included in the board, we model a baseline with 8 GB of LPDDR4 DRAM memory.

Our heterogeneous platform contains the same CPU and DRAM memory. However, instead of the GPU, we include the accelerators described in the previous sections. Table 4.5 contains the configuration of the DNN accelerator. After testing different configurations, we finally configured Tn, the parameter that scales the buffers and the NFU, to 16. This configuration provides enough

Table 4.6: Parameters for the Viterbi accelerator

Technology	28 nm
Frequency	600 MHz
State Cache	128 KB, 4-way, 64 bytes/line
Arc Cache	256 KB, 4-way, 64 bytes/line
Token Cache	128 KB, 2-way, 64 bytes/line
Acoustic Likelihood Buffer	64 KB
Hash Table	768 KB, $32K$ entries
Memory Controller	32 in-flight requests
State Issuer	8 in-flight request
Arc Issuer	8 in-flight request
Token Controller	32 in-flight request
Acoustic Likelihood Issuer	1 in-flight arc
Likelihood Evaluation Unit	4 FP adders, 2 FP comparators

performance to execute the TDNN system in real time with very low area and power overhead over the complete system. In order to reduce the size of the accelerator, we shrank the arithmetic units from 16-bit to 8-bit. Additionally, we reduced the clock frequency from the $0.98GHz$ specified in the DianNao paper to $55MHz$, which removes the need to segment the NFU pipeline stages. This clock frequency is enough for real-time execution of the TDNN system while largely reducing the power and memory bandwidth requirements, making the solution more amenable for low-power mobile systems. We estimate the area of the DNN accelerator to be $0.3mm^2$ using 28 nm technology nodes.

The Viterbi accelerator is configured as shown in Table 4.6. Compared to [120], we shrank the size of the caches and hash tables, reducing the area from $24mm^2$ to $3.34mm^2$. This large reduction in on-chip memory has a small impact on performance since the decoding graph included in the ASR system is significantly smaller than the one used in [120]. Note that the accelerator in [120] was designed for a single-pass ASR system that includes a more complex decoding graph for Viterbi search. However, our ASR system includes a rescoring pass with an TDNN-LSTM language model after the Viterbi search. Due to this rescoring pass, the decoding graph in the TDNN system can be largely reduced while maintaining accuracy, even though it includes a larger vocabulary. Specifically, the graph in [120] contains a vocabulary of 125k words and has a size of $618MB$ whereas the decoding graph of the TDNN system contains 200k words and only occupy $181MB$.

To evaluate the performance of the different components of the ASR system, we retrieve information from several sources. Measurements of energy and execution time during the execution in CPU were obtained from hardware performance counters. The DNN and Viterbi accelerators are modelled with cycle-accurate simulators in order to obtain the execution time of the ASR components executed in them and the activity factors of the different hardware components. To

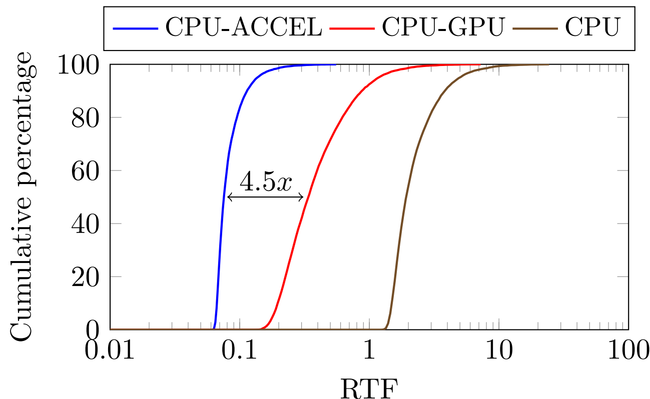


Figure 4.4: Cumulative distribution of the RTF for all utterances in the test set. The plots correspond to the execution of the kaldi system on the 3 hardware architectures: CPU, CPU-GPU and CPU-ACCEL.

estimate power and maximum clock cycle of the DNN and Viterbi accelerators, we relied on several tools. First, we used CACTI to estimate area, energy consumption and access time for the on-chip memories of the accelerators. Second, we implemented the different pipeline components in Verilog and synthesized them using Synopsys Design Compiler. The maximum frequency was set according to the minimum time required to propagate the signal through the logic components and memories, as reported by Synopsys Design Compiler and CACTI, respectively. To estimate total energy consumption, we obtained the activity factors from the cycle-level simulators and the energy cost of each operation and memory access from Synopsys Design Compiler and CACTI. The 8GB DRAM memory was modelled using *Micron TN5301_LPDDR4 System Power Calculator* [76] with the parameters from Micron's Z91M package.

4.3.1 Execution Time

We estimate execution time and *Real Time Factor* (RTF) by computing the time required to execute each component of the ASR system on the hardware it is mapped to. All the parts mapped to the CPU were measured directly by internal counters. Parts of the ASR system mapped to the accelerators were simulated to obtain cycle count, and then the number of cycles was multiplied by the cycle time of the specific accelerator to obtain execution time.

To analyze the gains achieved by using custom hardware, three alternative architectures were studied and compared. Figure 4.4 shows the RTF distribution among the utterances, plotted as cumulative frequency. *CPU* and *CPU-GPU* are the baseline platforms. *CPU* refers to executing all the processes on the CPU, whereas *CPU-GPU* executes all the highly parallel computations on the GPU (TDNN and RNN). *CPU-ACCEL* refers to our platform. It consists of the same CPU as the baseline platform, but, instead of the GPU, we include the aforementioned accelerators and execute the ASR system as described in section 7.1. The x axis is the RTF, whereas the y axis is the percentage of utterances decoded on at the specific RTF or lower. In RTF, lower is better. An RTF value lower than 1 means that the process is executed faster than real-time. We can draw two main conclusions from Figure 4.4. First, using custom hardware provides an important performance improvement (around 4.5x improvement compared to the GPU based system), and it is key to

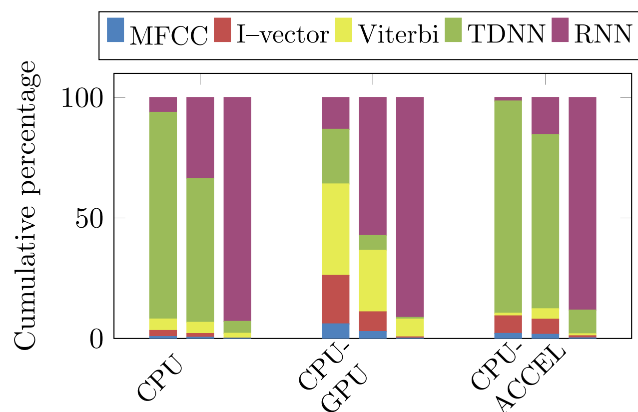


Figure 4.5: Execution time breakdown for the different ASR components obtained on the 3 hardware architectures. For each architecture, the three bars represent the utterances at percentiles 0, 50 *and* 100 *in the RTF plot (Figure 4.4).*

efficiently guarantee real-time for all utterances. Second, most of the utterances lay in a narrow region of RTF, especially for the *CPU-ACCEL* system, with some important outliers (around 10% of the test utterances in our experiments) laying very far from that region. A direct consequence of this high variability is that a system dimensioned to guarantee a specific performance for the worst case would be highly oversized for most of the utterances. However, these outliers represent real scenarios that cannot be ignored.

Our heterogeneous platform meets the real-time constraints for all the utterances in the Librispeech test set (more than 2k utterances and more than 5 hours of speech), achieving real-time factors of about 0.05x RT on average and 0.09x RT in the worst case.

In order to study the bottlenecks in execution time, we chose several utterances that represent different percentiles of RTF. More specifically, we sorted the utterances in ascending order of RTF and chose those located at the percentile 0, 50 and 100. Figure 4.5 shows the breakdown of execution time by ASR component for the three representative utterances. Additionally, the figure shows those three utterances executed on the baseline hardware. For each hardware platform, the bars represent the three selected utterances at percentile 0, 50 and 100. This figure clearly explains the large variability of execution time. The execution time of the utterances close to the 100% percentile is by far dominated by the TDNN-LSTM LM rescoring phase, which is also computed on the DNN accelerator. By looking at the lattice generated after decoding those utterances, we see that the lattice obtained after decoding the utterance at the percentile 100% occupies $281KB$ and requires 1507 RNN evaluations, whereas the lattice for the utterance at 0% occupies only 326 B and requires just 1 RNN evaluation. Utterances like those near the percentile 100% are very challenging to decode. Because of that, the Viterbi search has to explore a larger number of alternative paths, generating a larger lattice. The difficulty at decoding an utterance has an important impact on LM rescoring, which makes this component the main source of the RTF variability as shown in Figure 4.4.

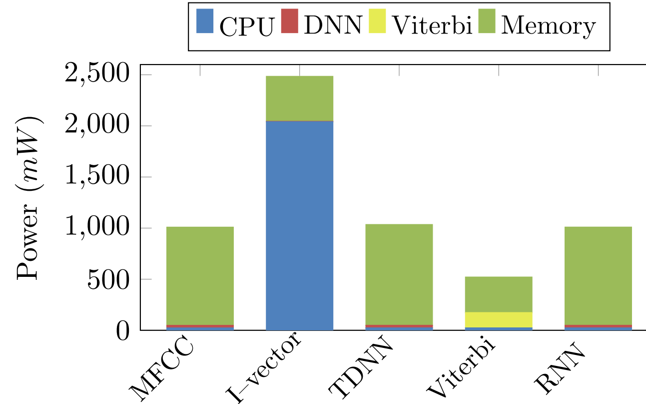


Figure 4.6: Power dissipation during the computation of the ASR components. Each bar represents the power dissipated by each ASR component, broken down by hardware subsystem.

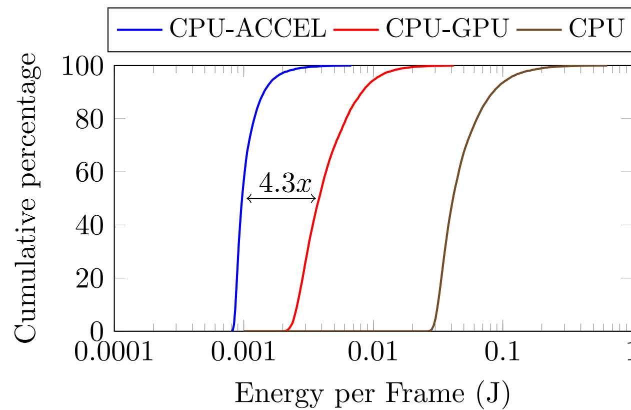


Figure 4.7: Cumulative distribution of the energy per frame for decoding for all utterances in the test set. The plot shows the energy for the execution on the 3 hardware platforms: CPU, CPU-GPU and CPU-ACCEL.

4.3.2 Power Consumption

Figure 4.6 shows the average power dissipated by the ASR system broken down into the different components of the TDNN ASR system. As we can see, the peak power is very close to $2.5W$, reached during the computation of the i–vector. Note that computing the i–vector requires the use of the CPU, the most power demanding hardware component in our system, whereas for the rest of the time the CPU is mostly idle. During the Viterbi computation, the DNN accelerator is power gated, so the power shown in the bar includes the power dissipated by the Viterbi accelerator, the CPU in idle mode and the main memory, resulting in $519mW$. The rest of the components, i.e. MFCC, TDNN and RNNLM, are computed almost exclusively on the DNN accelerator, while the Viterbi accelerator is power-gated and the CPU is idle, so the corresponding bars show the power dissipated by the DNN accelerator, the CPU in idle mode and the memory. The average power is slightly above $1W$, where most of it (about 95%) is due to the main memory, which is intensively accessed (4.1.2).

Regarding the energy consumption per utterance, our estimations show that decoding in our platform requires 4.3x less energy per frame than using the CPU-GPU system (Figure 4.7). As shown in Figure 5.1, most of the energy (71.3%) consumed by our platform is due to main memory. A 26.5% is consumed by the CPU, and the rest (less than 3%) is consumed by the accelerators.

All those results show that the proposed ASR system can be integrated into low-power devices due to its low area and power budget.

5 Leverage Run-time Beam Search Confidence

In the previous chapter, we describe a low-power heterogeneous platform designed to perform highly accurate ASR in real-time. In this chapter, we explore a mechanism to perform DNN inference using 4-bit arithmetic precision to further increase performance and energy efficiency during ASR decoding. We observe that some frames are more resilient than others, and consequently, we can decode those resilient frames with a 4-bit AM DNN without causing a severe impact on WER and then use the baseline 8-bit AM DNN to decode the rest of them.

5.1 Analysis of Bottlenecks

In this section, we analyze the main performance and energy bottlenecks of the Kaldi ASR system when it is executed in the hardware platform described in the previous chapter.

As described in the previous chapter, the ASR system is a Hybrid DNN-HMM system. It uses MFCC features enhanced with i-vectors for speaker adaptation. The AM is a TDNN network and the decoder is an HCLG graph built from a 200k word vocabulary and a pruned 3-gram language model, along with an HMM. The AM TDNN is quantized to 8 bits. The hardware platform consists of a low-power ARM CPU and 8GB of LPDDR4 DRAM complemented with two accelerators: one to perform DNN inference and another to perform Viterbi Beam Search.

5.1.1 Energy Bottleneck

Figure 5.1 shows the breakdown of energy consumed by each of the components of the hardware platform. Most of the energy (85%) is consumed by the DRAM memory. The rest of the

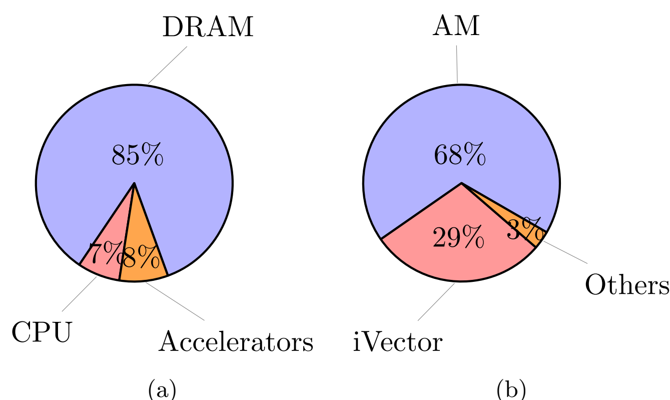


Figure 5.1: Breakdown for energy consumption during ASR evaluation on the mobile SoC presented in Section 4. Chart (a) shows the energy breakdown among hardware components during the AM evaluation. Here, it can be seen how reads and writes from the DRAM are responsible for most of the consumed energy, whereas chart (b) shows the energy breakdown by ASR component, where the clear bottleneck is the Acoustic Model TDNN evaluation, whereas

energy (about 15% of the total energy consumption) is consumed by the CPU and the accelerators. Chart 5.1(b) shows the breakdown of the energy consumed during the execution of each of the ASR components. The most expensive parts are the acoustic model and the i-vector computation, which account for 68% and 29% of the total energy consumed, respectively. The acoustic model is entirely executed in the DNN accelerator whereas the i-vector is mostly executed in the CPU with some parts offloaded to the DNN accelerator. We determined that the reason the AM consumes so much energy is that it is responsible for the majority of the DRAM traffic. Even though the DNN models are much smaller than other data structures, such as the embedding table and the decoding graph, the DNN models have to be read entirely many times per second whereas the other structures are only sparsely accessed. DRAM accesses during AM DNN inference account for 58.1% of the total energy consumed. Furthermore, we have identified that 99% of the memory accesses during TDNN evaluation are for reading DNN weights.

5.1.2 Performance Bottleneck

Regarding the execution time, AM DNN inference is also the main bottleneck as it takes 82% of the execution time. The *Feature Extraction* and the *Beam Search* account each for 14.8% and 3.2% of the execution time, respectively.

5.1.3 Optimize DNN inference

It is clear from the previous results that accessing the AM DNN is the main performance and energy bottleneck. The issue is that the DNN is too large to be kept inside the SRAM memory of the DNN accelerator, and thus has to be read many times per second from DRAM memory.

A well-known optimization to alleviate this bottleneck consist of pruning the network by an

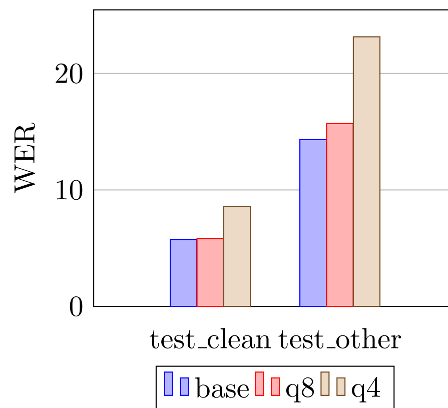


Figure 5.2: Comparison of the WER loss respect to the non-quantized model for various levels of quantization. While quantizing to 8 bits has a small impact on WER, more aggressive quantization sensibly degrades accuracy

iterative process of removing some weights and retraining. This approach results in a sparse network, which can be significantly smaller than the dense network. However, the pruning algorithm is expensive, and the inference with a sparse network requires important changes in the DNN accelerator.

Another well-known technique to alleviate this bottleneck is aggressive DNN quantization. However, quantizing to less than 8 bits results in an important degradation in recognition accuracy, making it a bad solution. Figure 5.2 shows the WER for *test_clean* and *test_other* evaluated with the TDNN acoustic model at different levels of quantization. While 8 bits results in minor accuracy loss compared to full precision, going to 4 bits increases the WER by 49% in test_clean and 61% in test_other, and if the weights are quantized to 2 bits, the system does not work, generating invalid transcriptions.

In this work, we explore decoding with a 4 bit quantized AM DNN. However, in order to minimize the negative effect on transcription accuracy, we follow a dynamic approach where we employ the 4-bit DNN only for selected frames, whereas the rest are decoded using the 8 bit quantized AM DNN.

5.2 Dynamic DNN Precision

As discussed in Section 5.1, fetching the DNN weights from main memory takes a large percentage of the total energy consumption. Quantizing the DNN is an effective technique to reduce the energy consumption derived from DRAM accesses. However, reducing DNN precision below 8 bits results in a significant accuracy loss. Therefore, we take a different approach and propose to quantize the AM DNN at two different levels of quantization: 8 bits and 4 bits.

However, determining in advance which frames require more or less precision is a challenging problem. In this work, we propose to look at the number of hypotheses expanded during the Beam

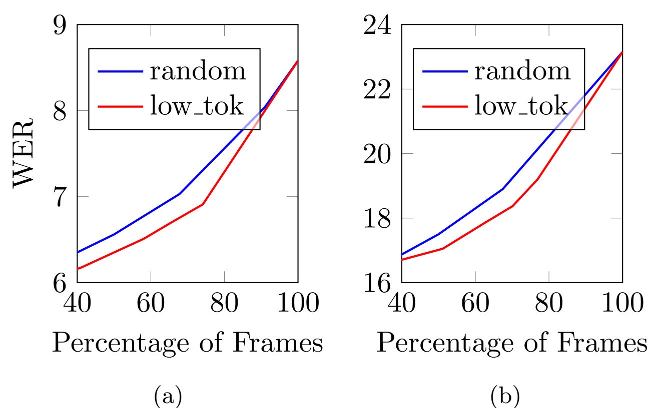


Figure 5.3: WER for a varying percentage of frames evaluated at low precision for a) test_clean and b) test_other. The curves represent the cases when the frames for low-precision evaluation are those with less number of tokens (high confidence), and when they are chosen randomly.

Search. Figure 5.3 shows the relation between the WER and the percentage of frames decoded using the 4 bit AM DNN. The x axis shows the percentage of frames decoded with the 4 bit AM DNN (the rest are decoded with the 8 bit AM DNN), whereas the y axis shows the resulting WER after decoding the benchmark with the related percentage of frames decoded with the 4 bit AM DNN. The blue line (random) shows the WER obtained when the 4 bit AM DNN is used at random times, whereas for the red line (low_tok), we chose the 4 bit AM DNN when there are not many expanded hypotheses. For that purpose, the number of hypotheses expanded during each decoding step is tracked and compared to a pre-set threshold.

According to our results, using the 4 bit AM DNN when there are not many hypotheses in the system seems to have a smaller effect on WER. Consequently, we can leverage the advantages of using an extremely quantized AM DNN while minimizing the effect on WER if we use the 4 bit AM DNN exclusively when there are few expanded hypotheses. When there are few hypotheses, we say that the decoder has *high confidence*. When there are many hypotheses, we say that the decoder has *low confidence* and call the following input frames *low confidence frames*.

We detect low confidence frames by keeping track of the number of hypotheses expanded during each decoding step and comparing it with a threshold. Setting this threshold, however, is not trivial. A simple option would be to track the number of hypotheses expanded during the decoding of the train set used to train the ASR model and set the threshold according to the desired ratio. However, we have observed that the threshold obtained with this method does not result in the desired ratio of low confidence frames when decoding a benchmark set. Figure 5.4 shows the ratio of frames classified as low confidence frames when using different static thresholds. To obtain the threshold values, the train set was entirely decoded using the 8-bit AM DNN, then we sorted the frames according to the number of hypotheses expanded by Beam Search and chose the number of hypotheses at the 30, 50 and 70 percentiles. The number of hypotheses expanded at those percentiles was set as threshold and then we decoded the test_clean and test_other sets with the different thresholds. The ratio of frames classified as low confidence by the thresholds in the test sets does not match the expectation by a wide margin, implying that this approach to set

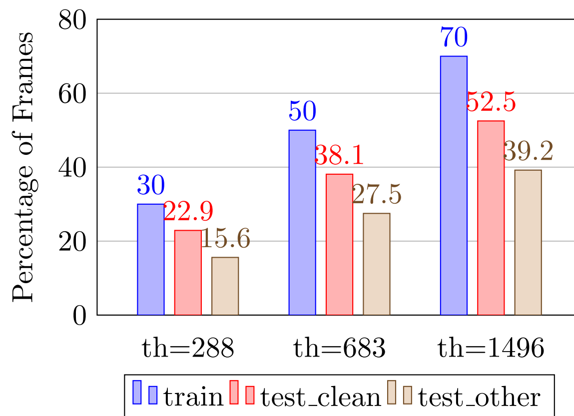


Figure 5.4: This plot shows the ratio of frames classified for low precision computation in our technique when using a fixed threshold. To obtain this threshold, the train set is evaluated at high precision. Then, all the frames are sorted according to the number of hypotheses expanded during Beam Search. The threshold value is chosen from the number of hypotheses expanded at different percentiles. As we can see, the desired ratio is not achieved in the test sets.

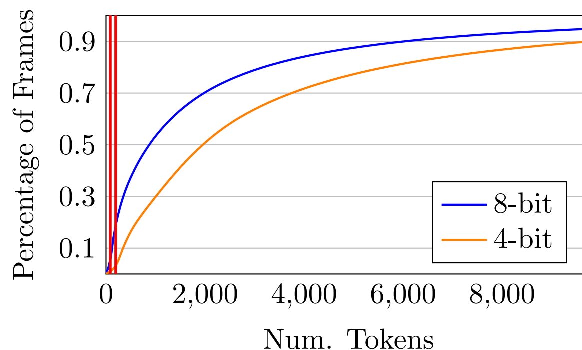


Figure 5.5: When low precision is used, the number of tokens expanded during Beam Search generally increases.

the threshold is not very reliable. This may be because a frame decoded with the 4 bit AM DNN will likely result in different hypothesis scores and even different hypotheses being expanded, which affects the behaviour of the beam search during the decoding of the subsequent frames.

We observed that, generally, using an aggressively quantized AM DNN result in more hypotheses being expanded. In other words, it reduces the overall confidence of the decoder, which is consistent with the observation made by Yazdani et. al [119]. Figure 5.5 shows the cumulative frequency of frames from test_clear and test_other according to the number of expanded hypotheses when the DNN is computed always at 8-bits and when it is always computed at 4-bits. In this case, the percentile 50 for the test set evaluated at 8-bits is at 883 tokens, whereas when the test set is evaluated at 4-bits, it is at 1961 tokens. It is clear from the figure that when the acoustic model is evaluated at lower precision, the overall number of hypotheses expanded per frame increases.

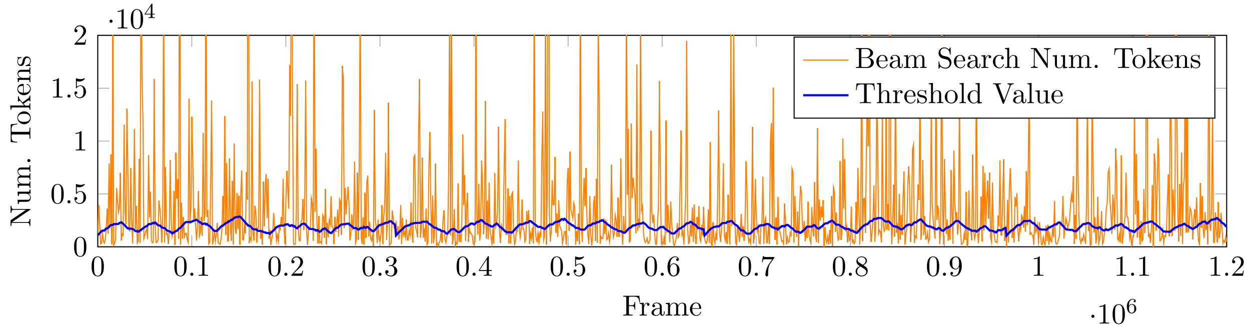


Figure 5.6: Threshold value computed by the proposed heuristic compared with the number of tokens expanded by Beam Search. The plot shows a span of 1.2 *million frames at* $\frac{1}{1000}$ *sampling rate.*

5.2.1 Dynamic Threshold Computation

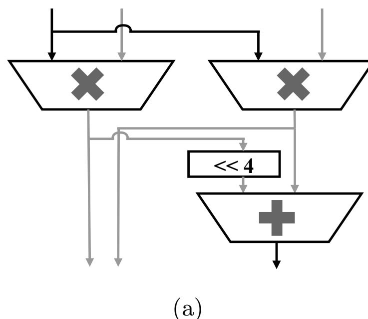
We propose an algorithm to perform small run-time adjustments to the threshold. This algorithm aims at 50% of the frames classified as low confidence frames. For that, we keep a variable, h, that contains the difference between the number of frames classified as low confidence and those classified as high confidence. The heuristic tries to keep h at 0 by increasing or decreasing the threshold when h is higher or lower than 0, respectively.

To avoid the threshold from oscillating wildly, we define an additional variable, h_l, which also contains the difference between frames classified as high and low confidence, but it is constrained to a window of the latest frames. This tells us if h is increasing or decreasing, as well as the approximated speed at which it is changing. If $h_l > 0$, we assume that the number of frames at low precision is increasing. If the opposite is true, i.e. $h_l < 0$, we assume that the number of frames evaluated at low precision is decreasing. With these values, we update the threshold (Th) using the following formula:

$$Th = \begin{cases} Th - \Delta & h > 0 \& h_l > 0 \\ Th + \Delta & h < 0 \& h_l < 0 \end{cases} \tag{5.1}$$

When the system has classified more frames as high confidence than low confidence and the tendency goes towards increasing high confidence frames, the heuristic decreases the threshold by Δ, so it is more difficult for frames to be classified as high confidence frames. On the other hand, if it has classified fewer frames as high confidence frames, both globally and in the local window, the threshold is increased, so more frames are classified as high confidence. Both Δ and the starting value of Th are parameters of the system. To avoid using floating-point arithmetic, Δ is an integer value, and we introduce another parameter to regulate the number of frames between consequent threshold updates.

This heuristic means that the threshold is going to fluctuate. To put the amplitude of the threshold oscillations into perspective, we can compare it with the number of hypotheses expanded per frame (Figure 5.6). Since the oscillations in both values are orders of magnitude apart, we can



(a)

Figure 5.7: Schematic of the multiplier unit included in our design. Light grey arrows represent half-precision values. This unit receives two full-precision values, which are interpreted as one full-precision and two half-precision values when operating in half-precision mode.

conclude that the proposed heuristic leads to a well stable threshold.

5.2.2 Changes to the DNN Accelerator

To implement the described technique, we quantize the weights from the full-precision Acoustic Model into two levels (8-bit and 4-bit), which we keep stored in memory. On each frame, depending on the number of hypotheses expanded by the Beam Search during the previous decoding step, we command the DNN accelerator to use either the 8 bit or the 4 bit AM DNN.

The 4-bit model is half the size of the 8-bit model, which results in half the time required to read the model from the main memory while using the same bandwidth between the accelerator and the DRAM memory. To take advantage of that in the most efficient way, we modified the accelerator so it can perform computations in base precision or half-precision with the same hardware, so we can double the number of operations per cycle when operating in 4-bit mode, with small area and power overheads over the baseline design.

The DNN accelerator now has to support two different modes of operation: *base-precision* and *half-precision*, which in this case means operating at 8 bits or 4 bits.

One of the main parameters of the accelerator is Tn, which configures the number of NFUs and the NFU vector size. In the baseline design, each NFU carries out the computation of a different neuron, whereas the NFU vector size enables to compute in parallel several inputs for that neuron.

Without loss of generality, we assume a configuration of the DNN accelerator with $Tn = 16$, using 8-bit weights for base-precision mode and 4-bit weights for half-precision mode. Hence, during the full-precision mode, the DNN accelerator computes 16 neurons in parallel, and for each of those, 16 parallel inputs. To do so, it receives 16×16 weights and 16 inputs each cycle.

For half-precision mode, however, the accelerator receives 32×16 weights and 16 inputs per cycle (i.e., 32 neurons are computed in parallel), that is, the size of the input buffer is the same as in the baseline design. We decided not to quantize the inputs to half-precision because we found it has an important impact on WER for a small benefit on performance. During half-precision

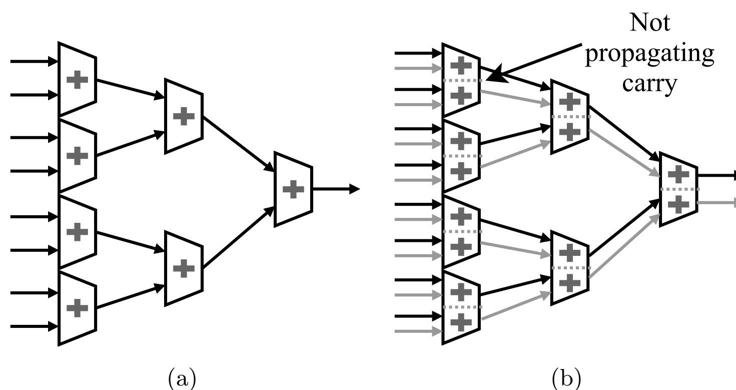


Figure 5.8: Schematic of a basic add-tree (5.8(a)) and our duplex add-tree when operating in half-precision mode (5.8(b)). In the latter, each arrow represents a half-precision value.

mode, we partition each CU so it computes 16 inputs for 2×16 neurons. In this case, each CU receives 2×16 half-precision weights and 16 base-precision inputs. In this solution, we still have to modify the multiplication units to support both base-precision (one 8-bit×8-bit multiplication) or half-precision (two 8×4 multiplications). However, since in half-precision mode we are computing two neurons at the same time at each compute unit, every two multiplications share the same 8-bit input operand. We design our multiplication units to support both the multiplication of two base-precision operands or two half-precision operands multiplied by the same base-precision operand. Figure 5.7 shows a diagram of our multiplication unit.

Since in half-precision mode each CU accumulates two different neurons, the add-tree must be able to perform one base-precision accumulation of 16 values or two half-precision accumulations of 16 values each. For that purpose, we modify the adders in the tree so the transmission of the carry from one half to the other is conditioned on the mode of operation. By doing this simple modification, when the add-tree operates in half-precision mode, each adder operates as two independent adders, and thus the complete tree is unfolded in two separated trees, as shown in Figure 5.8.

Additionally, since we are merging different levels of precision, the bit-width of the adder units has to be carefully set to avoid arithmetic overflow.

The activation unit performs the neuron activation function after all the neuron inputs have been accumulated. Since this unit is only used at the end of the neuron evaluation, and the accelerator is alternating the computation of several neurons (to leverage temporal locality of inputs), there is plenty of time from one activation to the next, and thus the activation unit only requires support to serialize the output from the compute unit when operating in half-precision mode. A similar argument applies to the output buffer.

5.2.3 Changes to the Beam Search Accelerator

To compute the threshold updates following the proposed heuristic, we introduce some modifications in the Beam Search accelerator. First, it has to count the generated tokens on each search step and keep track of the threshold, Th, and the variables h and h_l used by the heuristic. To this end, we modify the *Token Issuer* to include a few adders, a very small buffer for the h_l window, and a register for each variable, resulting in a negligible area overhead.

With these modifications, the *Token Issuer* keeps track of the threshold and the number of tokens expanded during each Beam Search step. After each step, the number of expanded tokens is compared with the threshold, and the required precision is exposed to the DNN accelerator. When a new frame is captured and transformed into a *Feature Vector*, the DNN accelerator computes the inference for that frame in the required precision.

5.3 Experimental Results

In this section, we evaluate the speedups and energy savings achieved by our dynamic DNN precision scheme based on Beam Search confidence. The baseline system is the mobile SoC platform described in Chapter 4. It includes a multicore ARM CPU, a DNN accelerator and a Beam Search accelerator. We have implemented our scheme on top of this SoC as described in Section 5.2.

In order to implement our technique, the baseline accelerators require some modifications. More specifically, the DNN accelerator must support two operation modes: base-precision and half-precision, whereas the Beam Search accelerator has to compute the heuristic and keep track of the threshold. These modifications, made as described in Section 5.2, result in a negligible area overhead of 3.1% over the baseline accelerators. In the final setup, the DNN accelerator occupies an area of $0.42mm^2$, split between *buffers* (25.5%), *MULT arrays* (62.6%) and *AddTrees* (11.9%), whereas the Beam Search accelerator occupies $3.34mm^2$.

Since we keep an additional DNN model (AM quantized to 4 bits) in DRAM memory, our solution incurs a small memory footprint overhead. However, the additional model is fairly small, increasing the memory footprint by just 3.8%.

In the current setup, the average power of the complete system, including CPU, DRAM and the accelerators is $1.17W$ (3.3% increase over the baseline).

5.3.1 Performance Gains of Dynamic Precision AM

Since our heuristic modifies the threshold slowly, during a single utterance evaluation it does not change by a large extent, and thus, although for a long run the number of frames decoded with the 4 bit AM DNN converges to 50%, individual utterances are decoded with different ratios. Figure 5.9 shows the distribution of utterances according to the percentage of frames evaluated with the 4 bit AM DNN. Most of the utterances fall between $40 - 60\%$, but a few of them decode more than 80% of their frames using the 4 bit AM DNN, resulting in huge local performance gains

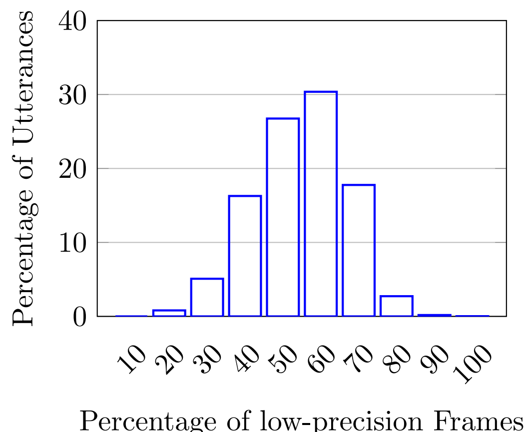


Figure 5.9: Frequency of utterances (vertical axis) grouped by the percentage of their frames computed at low precision (horizontal axis).

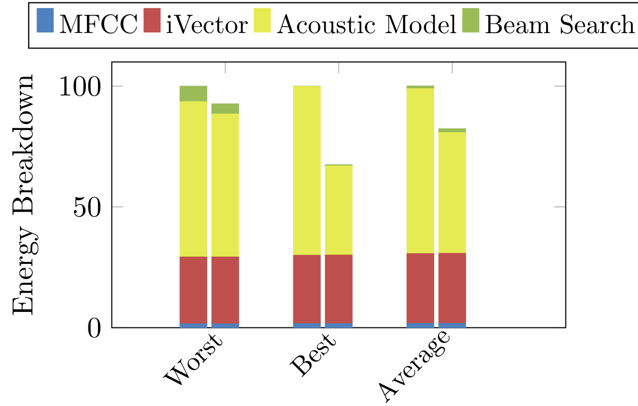
and energy savings.

Figure 5.10 shows the savings obtained from the proposed technique compared to the evaluation on the baseline platform. Both for energy and time, three cases are plotted: the *Worst Utterance*, the *Best Utterance* and the *Test Set Average*. The *Worst* and *Best* utterances are chosen regarding the percentage of frames evaluated at half-precision. We can see how even in the worst case, significant savings are achieved.

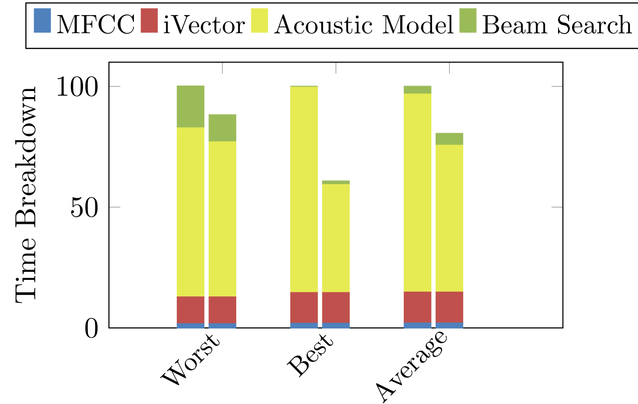
By applying our technique, we can save up to 47.2% of energy and reduce the execution time up to 47.4% for the Acoustic Model evaluation on utterances where the decoder confidence is high for most of its frames. On average, our scheme reduces energy consumption by 25.6% and execution time by 25.8% when evaluating the complete test set.

Since the 4 bit Acoustic Model DNN is half the size of the full-precision (8-bit weights) network, whenever a frame is evaluated at low-precision, we save half the reads from the main memory. Operating at half-precision results in significant speedups for two reasons. First, the Neural Function Units (NFUs) are modified so they can operate at double throughput in half-precision mode with negligible hardware overheads. Second, the DNN accelerator is memory bound since data reuse is largely limited in TDNN networks and, hence, reducing half of the reads from main memory results in large performance improvements. Therefore, the execution time for Acoustic Model evaluation is reduced by approximately one half during low-precision frames.

On the other hand, the reduction in energy consumption is also mostly explained by the reduction in reads from off-chip memory. As detailed in Section 5.1, off-chip reads of the acoustic model weights are the main bottleneck of the system, contributing to 85% of the energy consumed during Acoustic Model evaluation. Another source of energy savings comes from the reduction in static energy consumed by the rest of the components during the time that the Acoustic Model is being evaluated. Since around 50% of the total number of frames are evaluated at low precision, the observed savings of around 25% in time and energy during Acoustic Model evaluation are consistent.



(a)



(b)

Figure 5.10: Energy consumption (a) and execution time (b) normalized to the baseline. The bars in each plot represent: Worst Utterance, Best Utterance and Test Set Average for the baseline and proposed scheme respectively.

When we take into account the complete ASR system, the savings obtained in the acoustic model evaluation translate to an average reduction in energy consumption of 16.9% and a reduction of 19.5% in execution time (Figure 5.10). As discussed in Section 5.1, when the low precision acoustic model is employed, the average confidence of the Beam Search is decreased, which translates to a decrease in the performance of the Beam Search when our technique is used. Consequently, the energy and time consumed by the Beam Search are generally increased with respect to the baseline. However, even for the worst cases observed in the test sets, the benefits exceed the overheads, resulting in a net improvement in performance for all the utterances.

5.3.2 Effect on Accuracy

As discussed in Section 5.1, quantizing the AM DNN to 4 bits results in major degradation of transcription accuracy. However, by restricting the use of the 4 bit AM DNN to high confidence

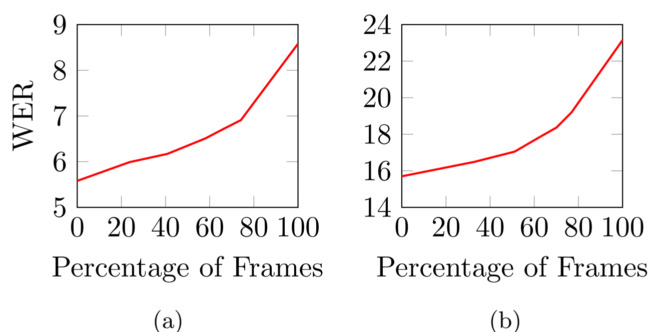


Figure 5.11: Sensitivity analysis of the percentage of frames set as target for low precision evaluation. The curves represent the WER obtained when some percentage of frames is evaluated in low precision for test_clean (a), and test_other (b).

frames, we managed to minimize the WER loss. Our experiments show that by using the heuristic proposed in section 5.2.1, we can use the 4 bit AM DNN to decode 50% of the frames incurring in less than 1% absolute WER loss for test_clean, and 1.35% for test_other.

In order to select a target for the percentage of frames computed at low precision, we performed a sensitivity analysis, modifying the target percentage from 0% (every frame in high precision) to 100% (every frame in low precision). Figure 5.11 shows the relation between WER and savings. Note that the time and energy savings are proportional to the percentage of frames evaluated at low precision. If some percentage of frames, $x\%$, is evaluated in low precision, we save around $x/2\%$ of time and energy during the DNN evaluation. The figure shows a curve with an elbow around $x = 50\%$ for both cases: test_clean, and test_other. Choosing this target grants a significant gain with negligible WER loss.

6

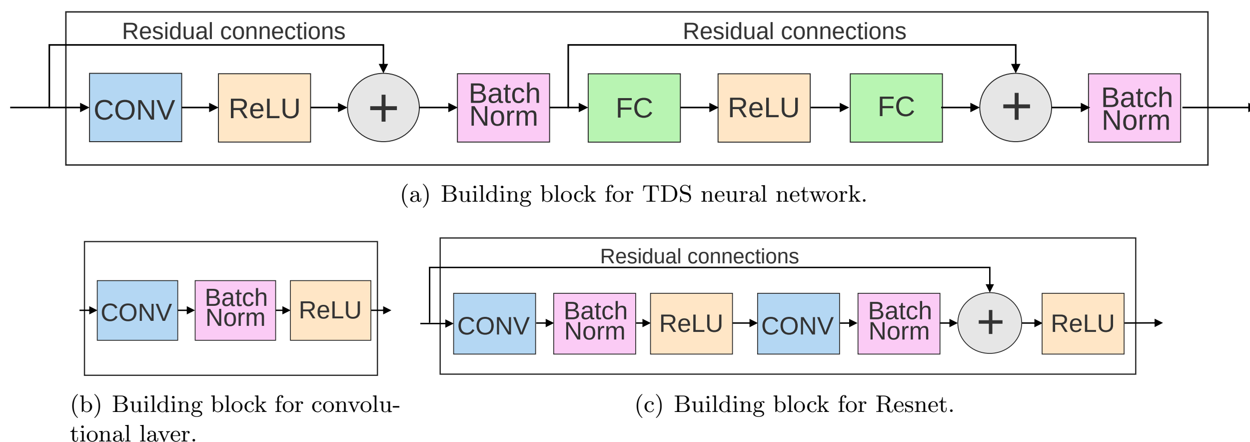
Predicting ReLU outputs to Skip DNN Computations

Hybrid HMM-DNN ASR systems such as the TDNN system studied in previous chapters obtain high-quality transcriptions by overcoming some of the limitations of HMM-GMM systems. However, their training process is complex and rely heavily on expert knowledge. End-to-end systems aspire to overcome those limitations by removing the HMM models and focusing on the DNN. These DNNs often contain ReLU activation functions. ReLUs generate abundant sparsity during run-time, creating huge opportunities for optimization. This chapter describes a technique to predict which ReLU-activated neurons will output a value of 0 before computing the dot product. Skipping these neurons during inference results in fewer data movements and computations, and thus reduces latency and energy consumption. This technique, which we call *Mixture-of-Rookies* combines two prediction components to leverage self-correlation and spatial correlation among neurons, resulting in a very efficient, highly accurate predictor.

6.1 TDS System

For this work, we focus on one of the end-to-end ASR systems that obtain state-of-the-art accuracy in the librispeech benchmark. Specifically, we focus on a variation of the system proposed by Pratap. et al [89]. Throughout this thesis, we call that system the *TDS system*. It contains a 156M parameter acoustic model DNN composed of TDS blocks. This acoustic model DNN requires 2 B MACS to decode each second of audio. Each input frame is an 80-dim MFCC vector computed from a 25 ms window of audio shifted by 10 ms for each consecutive frame. It is trained as a CTC DNN. The output of the DNN is a score distribution among 9997 labels. These labels contain word-pieces generated from the SentencePiece toolkit [63] and the CTC blank label.

Decoding is performed by traversing a lexicon tree, or *trie*, that contains all the combinations



(a) Building block for TDS neural network.

(b) Building block for convolutional layer.

(c) Building block for Resnet.

Figure 6.1: Building blocks for different DNNs.

of word-pieces to form the 200k words in the vocabulary. Every time the acoustic model DNN generates an output, the decoder expands the hypotheses, traversing the trie one step further for each of them. Each hypothesis also traverses an n-gram language model. Every time the decoder reaches a trie node that represents a word, it traverses the language model graph one step further and includes the LM score in the score of the hypothesis. To account for out-of-vocabulary words, every time the decoder reaches a non-word node in the trie, it generates a hypothesis that considers the partial word at the head of the hypothesis a potential out-of-vocabulary word and includes a score in the hypothesis that represents that posibility. Furthermore, in order to generate all the hypotheses in the CTC path, the decoder tries to append the blank label to each of the hypotheses, as well as the previous label, to account for repetitions, in addition to all the reachable labels in the trie.

6.2 ReLU Activations in DNNs

Earlier proposals [77, 70, 24] focused on GMM based recognizers, with *CMU's Sphinx* as an usual software baseline, and vocabularies with less than 100k words (e.g. 5k/20k-word Wall Street Journal, 64K-word Broadcast News,...). More recently, Tabani et. al. [107] proposed an accelerator for the *PocketSphinx* system, configured to ed by an FC layer.

Diagrams 6.1(b) and 6.1(c) show very common building blocks for CNNs. Particularly, diagram 6.1(c) describes the structure or *ResNet*, a popular CNN for image recognition tasks. In these cases, the ReLu layers are often preceded by a *batch normalization* function [53], which is computed as follows:

$$o = \frac{dotprod(\overrightarrow{weights}, \overrightarrow{inputs}) - \mu}{\sigma} * \gamma + \beta$$

where μ and σ are the mean and standard deviation of each dot product in the training dataset,

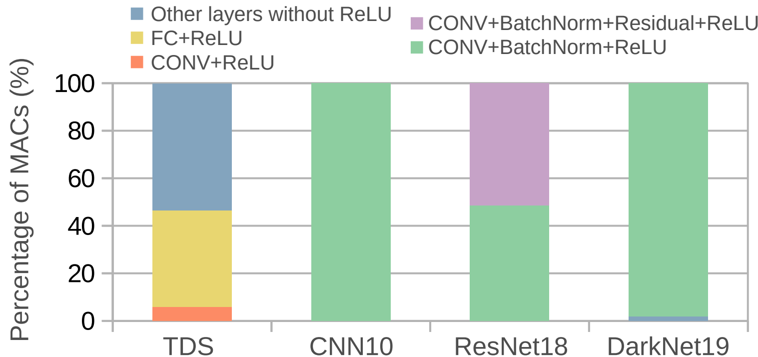


Figure 6.2: Percentage of MACs in each type of layer for a set of DNN aplications.

whereas γ and β are learnable parameters.

Additionally, the second ReLU in the *ResNet* block is preceded by a *residual connection* which adds together the output of a Batch normalization layer and the input of the block. In essence, we identify four major layer types: FC+ReLU, CONV+ReLU, CONV+BatchNorm+Residual+ReLU and CONV+BatchNorm+ReLU.

Figure 6.2 shows how the Multiply and Accumulate (MAC) operations are distributed according to the type of layer (FC+ReLU, CONV+ReLU, CONV+BatchNorm+Residual+ReLU and CONV+BatchNorm+ReLU) in different popular DNNs. In the TDS model for speech recognition, CONV and FC layers with ReLU represent 6% and 40% of the operations respectively, whereas, in *DarkNet19* and *CNN10*, two CNN for image recognition, more than 98% of the MACs are in CONV layers with batch normalization and ReLU. Finally, in *Resnet18*, CONV layers with batch normalization and ReLU represent 48% of the computations, whereas 52% of MACs are performed in CONV layers that also include residual connections. Therefore, to be widely applicable, a ReLU output predictor must support CONV and FC layers and provide accurate predictions in the presence of batch normalization and residual connections.

6.3 ReLU Output Predictor

In this work, we propose *Mixture-of-Rookies*, a technique to predict whether the input of ReLu is negative or positive. useful for neurons with a ReLU activation function, that is based on a combination of predictors with negligible overhead. This scheme exploits the synergies between different sources of information, improving prediction accuracy.

More specifically, we first leverage self-correlation by binarizing the DNN and using the binarized network to predict the outcome of the neurons, and then we exploit spatial correlation with a novel approach that generates clusters of neurons according to the angle between their weight vectors. An advantage of this approach is that the predictor is not based on any property specific to a class of DNNs, instead, it is a general technique applicable to a wide range of them.

Mixture-of-Rookies consists of 2 stages. First, an offline stage performs two tasks: a) profiles

the self-correlation of neurons, and b) generates clusters by grouping together neurons that share the same inputs and have the property that for any given input vector, either all the neurons in a cluster will produce a zero output or all of them will produce a non-zero output. Second, an online stage performs value prediction for the neurons during inference to avoid computing neurons whose ReLU activation function is predicted to produce a zero value.

Regarding the offline tasks, our technique employs a subset of training samples to perform a linear regression between binarized and base precision dot products in CONV and FC layers, obtaining a fitted line for each neuron. Besides, it groups the neurons of the same layer based on the similarity property described above and selects one neuron from each cluster to represent the whole group.

During DNN inference, *Mixture-of-Rookies* evaluates first the representative neuron for each group at base precision. If it generates a zero ReLU output, all the other neurons in the group are evaluated using 1-bit inputs and weights and the fitted line for each neuron is used to estimate the base precision ReLU output. If the estimated output of a neuron using this approach is also zero, then all the computations and memory accesses for this neuron are skipped and its output is set to zero. Otherwise, the neuron is computed using base precision. In other words, a neuron ReLU output will be predicted to be zero if and only if both prediction schemes agree on that. The next subsections provide further details on our predictor.

Leverage Self-correlation

The *Mixture-of-Rookies* predictor exploits the linear correlation between the ReLU input of a neuron computed in full precision and the ReLU input of a binarized version of the same neuron. We binarize neurons by simply taking the sign bit of its weights [25]. Exploiting this correlation, we build a predictor of each neuron's output by computing the dot-product of its binary weight and input vectors and predicting the dot-product of these vectors in full precision (i.e., the ReLU input) using this correlation. This approach has several advantages. First, the dot-product between 1-bit valued vectors does not require multipliers, simplifying the hardware to a large extent. Second, since the 1-bit weights are obtained from the sign bits of the full precision weights [100], they do not incur any memory footprint overhead since they do not have to be stored separately, but can be obtained directly from the full precision weights.

Figure 6.3 shows the ReLU inputs for a sample neuron from the TDS DNN in base (8-bit) precision (y-axis), versus the ReLU inputs for the binarized version of the neuron (x-axis). As it can be seen, there is a high linear correlation (correlation factor of 0.78). However, the sign of the ReLU input for 1-bit cannot be used as an estimation of the sign for the ReLU input in base precision, as a high linear correlation does not imply that the signs match. For example, points in the bottom-right quarter of Figure 6.3 are positive in the binarized version but negative in base precision. To mitigate this problem, we perform a linear regression and use the fitted line to obtain an estimated ReLU input from the binarized value. That is, we compute the coefficients of this fitted line and use it to transform the output of the binarized dot-product into the expected output of the base-precision dot-product.

Figure 6.4 shows the distribution of different levels of correlation among neurons for our bench-

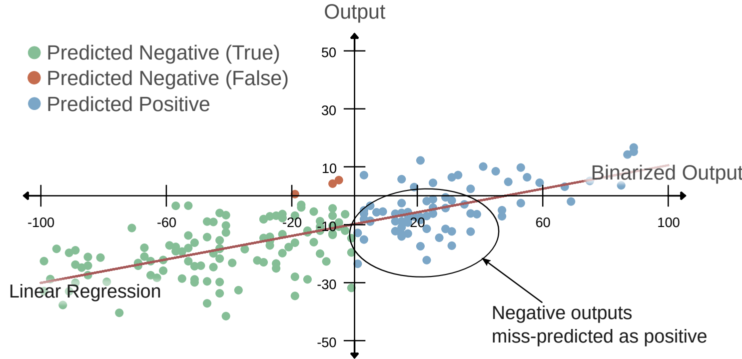


Figure 6.3: ReLU inputs for binarized neuron (x-axis) versus ReLU inputs for base precision neuron (y-axis).

marks. Even though most neurons exhibit a high correlation, a significant number of neurons have moderate or even low correlations. This observation is consistent with the observations made by Anderson et al. [13] and more recently, Silfa et al. [100]. A predictor based on 1-bit weights for a neuron with a low self-correlation coefficient is expected to make frequent mistakes, and consequently, reduce the overall accuracy of the DNN. Therefore, our predictor scheme is only enabled for neurons that show a high linear correlation with their binarized versions.

Mixture-of-Rookies performs a pre-processing stage on the trained model to extend each neuron's parameters with three additional ones: correlation coefficient (c), slope (m) and y-intercept (b) of the fitted line. These parameters are computed by using a randomly selected subset of the training dataset. Using this training subset, for each neuron, we obtain two series of data: ReLU inputs at 8-bit and 1-bit precision. We then compute the *Pearson correlation factor* (c) between the two series and perform a linear regression to obtain a fitted line $y = mx + b$. Parameters c, m and b are saved in the DNN together with the weights.

During DNN inference, each neuron is processed as follows. The correlation factor c is first fetched from memory. If c is lower than a threshold T, then the neuron is evaluated in base precision. Otherwise, the binarized dot product result, p_{bin}, is computed, and the fitted line is used to obtain the estimated base precision result $\hat{p}_{base} = m * p_{bin} + b$. If batch normalization and residual connections are used, then $\hat{p}_{base}$ is transformed by using the batch normalization parameters of the base precision neuron, and the residual input is added. If the resulting estimated ReLU input is negative, a zero ReLU output is predicted, skipping the evaluation of this neuron. Otherwise, the neuron is evaluated in base precision.

Since some neurons have low self-correlation with their 1-bit counterparts, using 1-bit predictors for the entire network will incur significant accuracy loss. Note that incorrectly predicting a ReLU output as zero will result in accuracy loss, as incorrect neuron outputs will be used, whereas incorrectly predicting an output as non-zero results represent a lost opportunity for saving computations but it has no impact on accuracy since in this case, the neuron is evaluated in base precision. To avoid accuracy loss, we leverage the aforementioned T threshold and only apply our prediction scheme for neurons whose correlation is higher than T. We use the training data to set appropriate values for T for each DNN, and verify its correctness using the unseen test data set. Note that T can be used to control the trade-off between computation savings and accuracy: the higher the

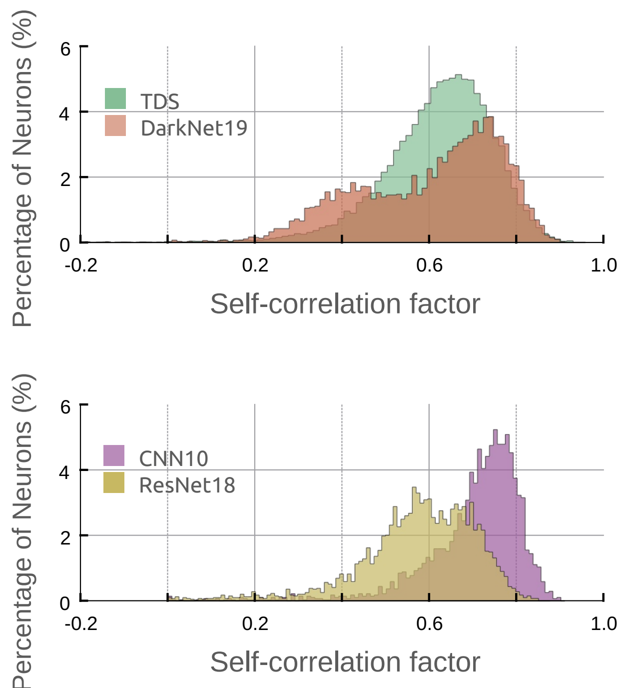


Figure 6.4: Distribution of neurons according to the Pearson correlation coefficient of the binary and base-precision ReLU inputs.

threshold the lower the accuracy loss but the smaller the savings.

Figure 6.5 shows the effect that different thresholds have on the accuracy loss and percentage of operations saved for our set of DNNs. Each line corresponds to a different DNN, and each point is obtained by using a different threshold T for linear correlation. The threshold is reduced from 1 (the first point on the left for each line) to 0.6 (the last point on the right). As it can be seen, the correlation threshold has a high impact on accuracy and percentage of savings. Furthermore, despite all the efforts to avoid incorrect predictions, the binarized predictor provides modest savings, 12% of computations for CNN10 and much less for the other networks, if accuracy loss is maintained. Lower thresholds result in larger savings, but at the cost of introducing a significant amount of errors, as the correlation between binarized and base precision neurons is lower. This study concludes that the binary predictor alone can provide very low benefits, and this motivates our proposal for a hybrid predictor.

Previous work proposed to use several bits [17], i.e. 4-bits, to improve self-correlation. However, we argue that 4-bits results in a significant overhead, and we propose in the next subsection an alternative solution that exploits spatial correlation to avoid mistakes done by the binarized predictor while incurring negligible overhead.

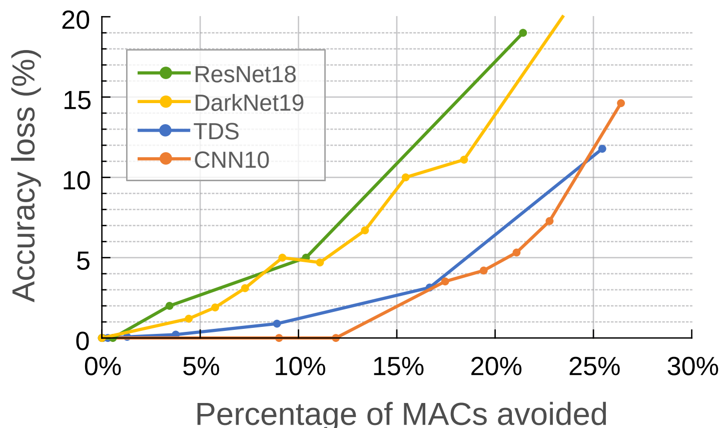


Figure 6.5: Effect of the correlation-based threshold on accuracy loss and percentage of operations saved for different DNNs.

Leverage Spatial Correlation

Our *Mixture-of-Rookies* predictor includes another scheme that exploits correlation among neurons with the same input vectors. This scheme aims to identify groups of neurons that share the same input vector and whose outputs are either all zero or all non-zero. In this manner, only one representative neuron of each group is evaluated during inference and, if it produces a zero output, the rest of the neurons of the group are assumed to produce a zero output without evaluating them. If the representative neuron produces a non-zero output, all neurons in the group are evaluated normally.

The key challenge is to identify a minimum set of groups with high zero/non-zero correlation among them. To this end, we analyze the relation between the angle of any two vectors to model the probability that the dot product between both of these vectors and a given third vector will result in values with the same sign. Since the sign of the dot product depends only on the angle between the operand vectors, we can assume a distribution for the third vector and model the probability of having same-sign results as a relation between said angle.

Given two vectors A and B, the dot-product between them is expressed as:

$$A \cdot B = |A| \times |B| \cos\theta \tag{6.1}$$

Where θ is the angle between A and B.

Since $|A|$ and $|B|$ are positive quantities, the sign of the result is given by the sign of $\cos\theta$, so it is entirely determined by θ. For convenience, we can limit the study to angles in the range $[0, 180]$ (θ is the small angle between the vectors) and conclude that the output will be negative only when θ is below 90°.

$$sign(\cos\theta) = \begin{cases} + & \text{if } \theta < 90^\circ \\ - & \text{if } \theta > 90^\circ \end{cases} \tag{6.2}$$

Note that the dot-product between 2 perpendicular vectors ($\theta = 90^\circ$) is 0, so its sign can be

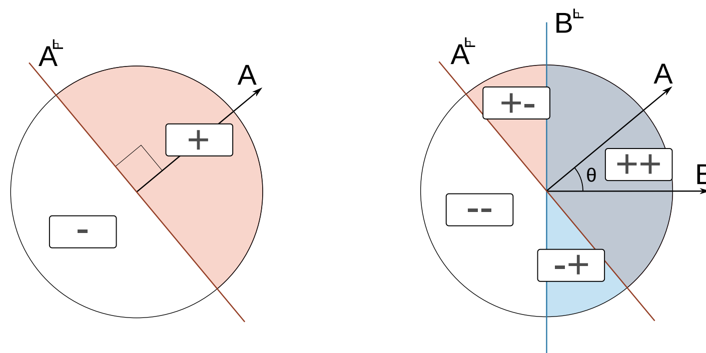


Figure 6.6: The line perpendicular to A partitions the circle into 2 sectors. Given a random C, the sign of $C \cdot A$ is determined by the partition in which C falls. If another vector B is added (right figure), the circle is partitioned into 4 sectors, which determine the signs of $C \cdot A$ and $C \cdot B$.

defined however is most convenient.

Figure 6.6 represents a circle and a vector A. If we draw a line perpendicular to A, the circle is divided into two halves. The dot-product between A and any vector from the half in which A is contained will result in a positive number. Correspondingly, the dot-product between A and any vector from the other half of the circle will result in a negative number. If we add a second vector B and its corresponding perpendicular line, the 4 regions (namely $R^{++}, R^{--}, R^{+-}, R^{-+}$) obtained by the overlapping of the 2 halves given by each vector characterize the range of vectors whose dot-products with A and B will result in each possible combination of signs $(++, --, +-, -+)$, and thus, it defines the probability of each possible outcome from $sign(C \cdot A)$ and $sign(C \cdot B)$ for a random vector C as the probability of C belonging to each of the previously defined regions.

Assuming that C follows a uniform distribution in the space (modeled as a hyper-sphere), these probabilities are given by the following expressions:

$$p(C \in R^{+-}|\theta) = \frac{\theta}{360} \tag{6.3}$$

$$p(C \in R^{-+}|\theta) = \frac{\theta}{360} \tag{6.4}$$

$$p(C \in R^{++}|\theta) = \frac{1}{2} - \frac{\theta}{360} \tag{6.5}$$

$$p(C \in R^{--}|\theta) = \frac{1}{2} - \frac{\theta}{360} \tag{6.6}$$

Where θ is the angle between the vectors A and B expressed as a degree magnitude between $0°$ and $180°$.

When this circle is expanded to a sphere, the area relationship between the regions (and hence the probabilities defined above) are preserved as a volume relation. We verified that this analysis holds for higher dimensions through a *Montecarlo* simulation.

If we use the dot-product between A (corresponding to the input weights of a neuron) and a random vector C (corresponding to an input vector) to predict the sign of the dot-product between

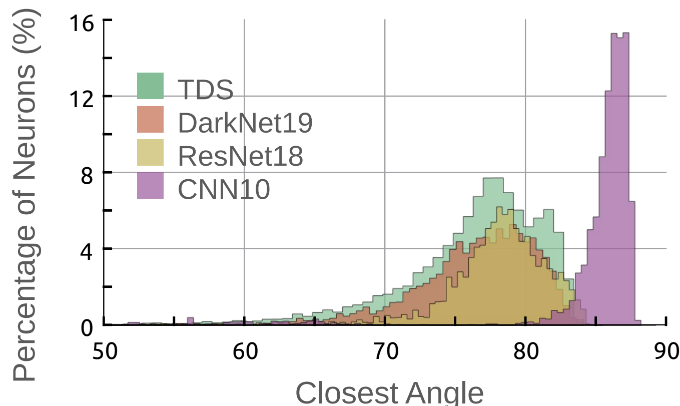


Figure 6.7: Distribution of angles between each neuron and its closest neuron.

B (corresponding to the weights of another neuron that uses the same input vector) and C, the worst case is when C is from the $-+$ region, because $A \cdot C$ will be negative, and thus $B \cdot C$ will be assumed to be negative, when in reality, $B \cdot C$ is positive. Since negative dot products result in a zero output when the ReLU activation function is applied, the output of the neuron with weights B will be wrongly assumed to be zero, without evaluating it (we call this scenario a false positive). The probability of a random C vector to be in the $-+$ region is given by expression 6.4.

As we can see, the probability of causing a false positive is 0 if the weights of the neurons are parallel, and increase up to 50% for perpendicular neurons.

Since neurons' weights are represented as very high dimensional vectors, if they were random vectors, we would expect them all to be almost perpendicular, meaning that if there are two neurons within a layer with a θ lower than $90°$, there is a certain degree of correlation among them. To measure the amount of spatial correlation in the TDS layers, we computed, for each layer, all the neuron-neuron angles and then, for each neuron, obtained the angle with its closest neuron (the neuron with which it has the smaller angle). Figure 6.7 shows the distribution of such angles. If there was no correlation between neurons, we would expect most of them to fall between $80°$ and $90°$. However, as we can see, the majority of angles fall between $70°$ and $80°$, and a significant number of them are even lower.

Based on the previous observations, we propose a negative ReLU input predictor that leverages the correlation existing between pairs of neurons separated by an angle lower than $90°$.

To leverage this property, we could cluster each neuron with its closest neuron. However, an algorithm that directly applies this clustering strategy will likely create problematic arrangements such as chains of associated neurons that will end up in the same cluster, but with neurons that are very far apart. Instead, we propose an algorithm that generates clusters of neurons around a principal neuron, which we call *proxy*, that will act as a predictor for the rest of the cluster members. This algorithm first generates a directed graph with the neurons as nodes and edges linking each neuron with its closest neuron. Then, the nodes are sorted by descending order of

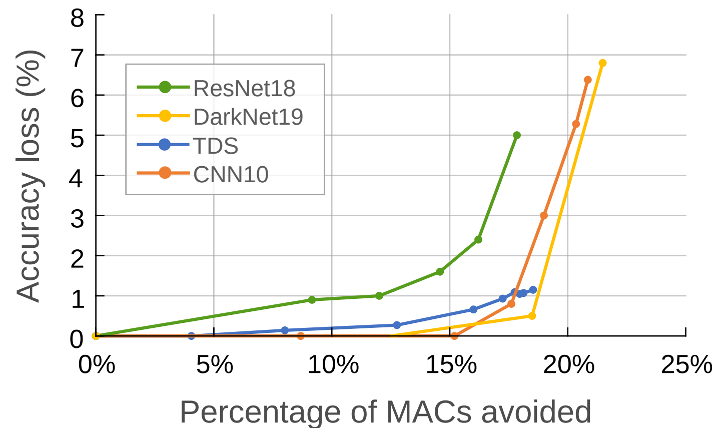


Figure 6.8: Accuracy loss versus percentage of computations avoided for the hybrid Mixture-of-Rookies predictor.

indegree (number of incident edges) and, starting from the node with higher indegree, the node is removed from the graph and included in the set of proxies, whereas all the nodes linked to it are removed, too, and included as members of the previous node's cluster. This process is iterated until there are no more nodes in the graph.

By looking at the distribution of closest angles among neurons (figure 6.7), it is clear that this technique alone will not result in good prediction accuracy. However, it provides useful information that can be leveraged to improve the performance of the self-correlation predictor. We combine this predictor with the self-correlation predictor described in Section 6.3. This predictor incurs negligible overhead since it only requires the neurons to be arranged in a specific way in memory (including an index value to re-arrange the outputs) and minor additional control logic.

Figure 6.8 shows how adding spatial correlation information improves the results of the ReLU output predictor. Compared to the binarized predictor in isolation, whose results are shown in Figure 6.5, the predictor that employs both self-correlation and spatial correlation achieves larger computation savings with small accuracy loss.

6.4 DNN Accelerator with ReLU Output Predictor

In this section we present a DNN accelerator that leverages our *Mixture-of-Rookies* predictor, described in Section 6.3, for energy-efficient DNN inference. Our accelerator is designed targeting use cases for inference in low-power devices, to support applications such as image or speech recognition on-edge. Hence, it is key to use very low area and power. Another constraining assumption for these use cases is that the input will be processed frame-by-frame (or image-by-image in the case of image recognition applications). For example, in [89], the authors explore very small dependency windows for the outputs of the TDS network presented in [46] in order to minimize word-to-transcription latency, which is desirable for many applications of speech recognition

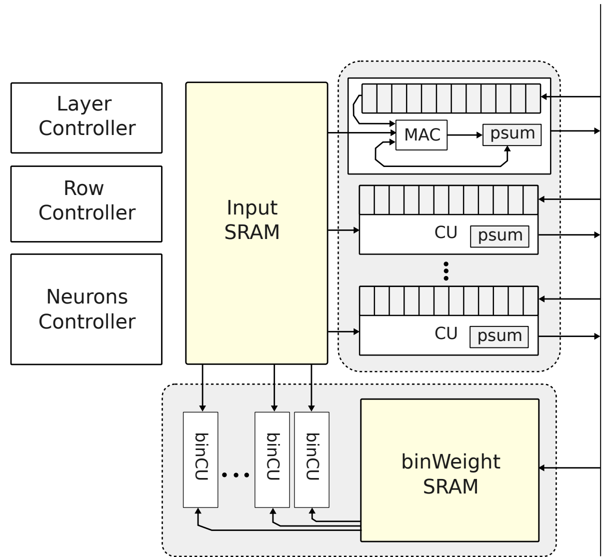


Figure 6.9: Accelerator with support for Mixture-of-Rookies ReLU output predictor.

on-edge.

Figure 6.9 illustrates the architecture of the accelerator. It contains three main control units: a *Layer Controller*, a *Row Controller* and a *Neurons Controller*, an SRAM memory to store the *inputs*, a set of *Compute Units (CUs)* to compute neurons and a *binary predictor* composed of an SRAM memory to store the *binary weights* and a set of *binary CUs* to compute the binarized neurons.

Each CU is responsible for the computations of the neurons assigned to it. The design has a configurable number of CUs, each with an interface to external memory. When a neuron is assigned to a CU, it generates requests to external memory and performs the required computation, accumulating the partial results on an internal register. To boost computations, the CUs perform in parallel several multiplications belonging to the same output. The number of parallel multipliers per CU is another parameter of the design.

6.4.1 Control Unit

The DNN processing is triggered by issuing an external request to the *Layer Controller*, which generates *Row Controller* requests to evaluate each layer of the DNN consecutively. The *Row Controller* divides the output of the layer in rows, and issues memory requests to load the required inputs to compute them. Since an output row will generally require many inputs, and consequently, a large input SRAM, the inputs for the row are divided in blocks, which are loaded sequentially. This allows us to keep the input SRAM small. To leverage inputs reuse in CNN shift-windows, the inputs are loaded taking CNN *stride* into account. Once an input window is loaded, the row controller issues a request to the *Neuron Controller*, which generates issues to the CUs and binCUs to compute the neurons for that input window.

Since our predictor creates dependencies between proxy neurons and their cluster members, we have to compute first the proxies to unlock the corresponding non-proxy neurons (see Section 6.3). Conceptually, the idea is to evaluate first all the proxies and generate a mask of neurons that are predicted to have a ReLU output of zero. Next, we evaluate the second predictor, binary predictor, for these neurons with predicted zero output and update the mask to include only those that are also predicted to produce a zero with the binary predictor. At this point, all the neurons not predicted to have zero ReLU output are assigned to CUs and their results are written back into memory.

The evaluation of the binary predictor can be overlapped with the evaluation of the proxies. As soon as a neuron is predicted to have zero ReLU output by the corresponding proxy, the neurons controller issues requests to the binary CUs to compute the prediction based on the binarized neuron. If binary predictor also indicates a zero output, the output of that neuron is predicted to be zero, and a 0 is written to external memory. Otherwise, the neuron is assigned to a free CU for full precision computation.

The computation of the proxy neurons does not generates any overhead in execution time since they would have to be computed anyway, and neither does the evaluation of the self-correlation binarized predictor, which is performed in parallel with the rest of the neurons.

In our hardware implementation, we do not store the entire mask in memory as we interleave the evaluation of proxies and non-proxy neurons. As soon a proxy is evaluated, the corresponding non-proxy neurons are assigned higher priority than proxies, meaning that as long as there are available non-proxy neurons, they will be assigned to any free CU. Only when there is none, the proxies are assigned to CUs. Note that we still require a small buffer to keep track of the available cluster members. However, this implementation provides two advantages: a) the buffer is smaller than the memory required to store the mask and b) it does not impose a maximum output size.

6.4.2 DNN Format

In order to support the execution flow previously described, we provide format to store the DNN in main memory, illustrated in Figure 6.10. The DNN layers are divided in two tables. The first table contains the proxy neurons. Each row contains an *index (idx)* field to indicate the original position of the neuron, a *cluster size* field to indicate the number of neurons in its cluster and finally, the weights of the neuron. The second table contains the non-proxy neurons sorted by the cluster they belong to. This means that the neurons associated with the first proxy in the proxy table occupy the first positions, the next positions are filled with the neurons associated with the second proxy and so on. Each row in this table contains the weights, the binary weights and an index to their original position. Since the binary weights are the sign bits from the weights, the sign bit is removed from every weight to offset the memory footprint overhead of storing the binary weights, and to avoid any increase in memory reads, which would otherwise affect the performance benefits.

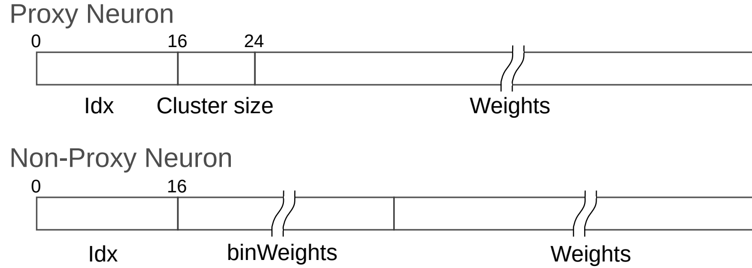


Figure 6.10: Format to store the DNN in external memory.

6.4.3 Compute Units

The accelerator contains a group of CUs to evaluate neurons. Each CU operates independently and is connected to main memory through its own ports. They are assigned neurons by the neurons controller and are responsible for the computation of that neuron. As soon as a CU receives a request to process a new neuron, it starts fetching its weights from external memory to the internal buffer. Next, it reads inputs sequentially from the input SRAM and performs the dot product between inputs and weights, whose result is temporarily stored in its *partial sum (psum)* register until the dot product is completed. The CUs have a design parameter to adjust the MAC unit width, which allows for computing several MACs in parallel. Note that the port width of the weight buffer is adjusted correspondingly.

6.4.4 Binary Prediction Unit

To support our prediction scheme, we include a *Binary Prediction Unit*. It is composed of an SRAM memory to store the binary weights for the non-proxy neurons and a set of binCUs. The binCUs are similar to the CUs, they process neurons and operate independently. However, they do not contain a weight buffer because they do not have to access external memory. Instead, they read binary inputs and binary weights from input memory and the binary weight SRAM, respectively, to perform the binary dot-product. Since these units perform binary multiplications and counting instead of a full MAC, their circuit is much simpler than the *CUs* [100].

6.5 Results

In this section, we present the speedups and energy savings achieved by the Mixture-of-Rookies predictor, when implemented on top of a DNN accelerator as described in Section 6.4. We compare our proposal with a baseline accelerator that does not include the *binWeights* SRAM and the *binCU* units, so the energy consumed by them is considered and reported as overhead for our predictor.

We first evaluate the accuracy of the proposed *Mixture-of-Rookies* predictor. Figure 6.11 shows the percentage of correct and incorrect predictions. "Correctly predicted zero" means that the predictor indicates that the ReLU output will be zero and it is correct. In this case, neuron evaluation is skipped, avoiding all the related computations and memory accesses for base precision

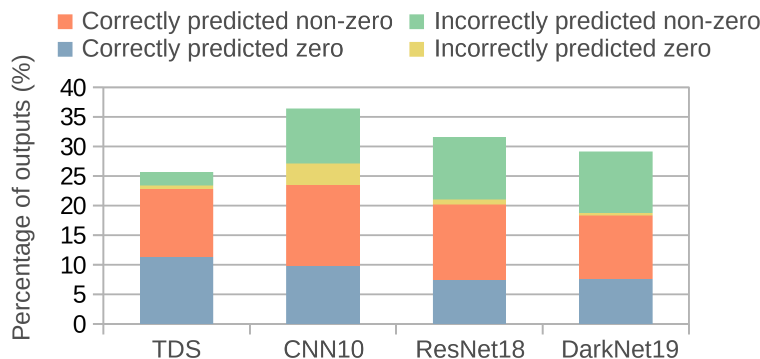
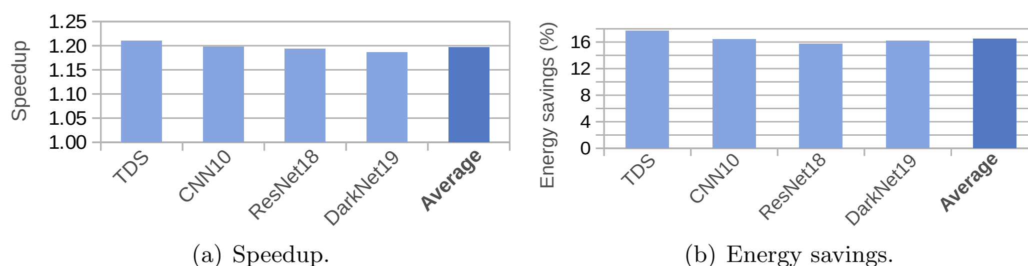


Figure 6.11: Percentage of outputs that are correctly and incorrectly predicted as zero or nonzero by our Mixture-of-Rookies predictor.



(a) Speedup.

(b) Energy savings.

Figure 6.12: Performance and energy savings achieved by our Mixture-of-Rookies ReLU output predictor compared to the baseline.

neuron, without affecting DNN classification accuracy. This is the case for 7%-11% of the outputs in our DNNs. “Incorrectly predicted zero” means that our scheme predicts a zero output, but the base precision output is non-zero. Neuron evaluation is also avoided, but these mispredictions may impact classification accuracy as they introduce errors in the DNN, since the output of these neurons is incorrectly set to zero. As it can be seen in Figure 6.11, this type of mispredictions are fairly infrequent: 0.65%, 0.8%, 0.4% and 3.6% for TDS, Resnet18, Darkent19 and CNN10 respectively. We verified that the impact on DNN accuracy due to these mispredictions is lower than 1% in our DNNs.

On the other hand, “incorrectly predicted nonzero” shows neurons where a non-zero output is predicted but the ReLU output is zero. These mispredictions have no impact on accuracy, as the neuron is evaluated when *Mixture-of-Rookies* predicts non-zero, but they represent a missed opportunity for saving computations. Finally, “correctly predicted nonzero” category shows that between 10% and 13% of the outputs are non-zero values correctly identified by our predictor. Note that the four categories shown in Figure 6.11 do not add 100% as there are neurons where the predictor is not applied for several reasons. First, our scheme is not applied in DNN layers that do not use ReLU activation function, this is common in TDS network. Second, *proxy* neurons, i.e. centroids, are always evaluated in our scheme. Third, our predictor is disabled for neurons that show poor linear correlation with their binarized versions.

Figure 6.12(a) shows the speedups achieved by our *Mixture-of-Rookies* predictor. Our prediction scheme provides consistent and significant performance improvements, providing 19.8%

speedup on average. These speedups are due to skipping neuron evaluation when the predictor indicates that its ReLU output will be zero. Note that this avoids both computation and memory accesses, since weights for those neurons do not have to be fetched from main memory. Computations related to the predictor itself are largely overlapped with useful computations and, hence, they do not introduce any performance penalty.

Regarding energy consumption, Figure 6.12(b) shows the energy savings achieved by our ReLU output predictor. As it can be seen, our technique obtains significant energy savings across all the applications, reducing energy by 16.5% on average. Our system improves energy consumption because the number of computations and memory accesses is reduced proportionally with the number of neurons skipped. Furthermore, the hardware for our *Mixture-of-Rookies* predictor represents a small overhead: 5.3% in area and less than 1% in energy consumption (included in the results). These overheads are more than offset by the benefits so the net result is an important reduction in energy consumption.

7

Programmable Low-Power Architecture for ASR

In the previous chapters, we proposed different techniques to improve performance and reduce energy consumption of ASR. Those proposals leverage specific properties of the ASR systems to increase performance and energy efficiency of the ASR systems. The different components of the ASR system are executed on very specific accelerators. However, ASR is a rich and fast-changing field and very specific accelerators risk becoming obsolete fast. In this chapter, we propose a programmable architecture for low-power ASR and demonstrate that it can run state-of-the-art ASR in real-time. The first section is a detailed description of the architecture, the second section is a case study where we show how the TDS system can be implemented in this platform and in the last section, we present our estimation on chip area, power and performance.

7.1 Architecture of ASRPU

Despite the differences among ASR systems, most of them follow a similar overall algorithm. We leverage that to design an accelerator that provides enough flexibility to support the differences between them while automating wherever possible to speed up the ASR process and simplify the software implementation.

This section provides a detailed description of the architecture of ASRPU. The accelerator (Figure 7.1) is divided into 3 major blocks: *Command decoder*, *Execution unit* and *Hypotheses unit*. The command decoder provides the interface to the accelerator via a set of commands. This includes commands to start an ASR decoding step, to finish decoding an utterance and to configure different parameters of the accelerator. The execution unit executes the program that implements the ASR system. This program is composed of small sub-programs, *kernels*, written by the ASR designer to implement each part of the ASR system. The execution unit contains a

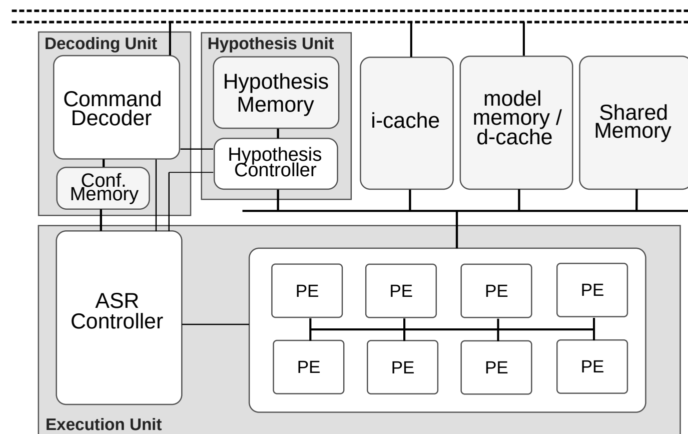


Figure 7.1: Architecture of ASRPU

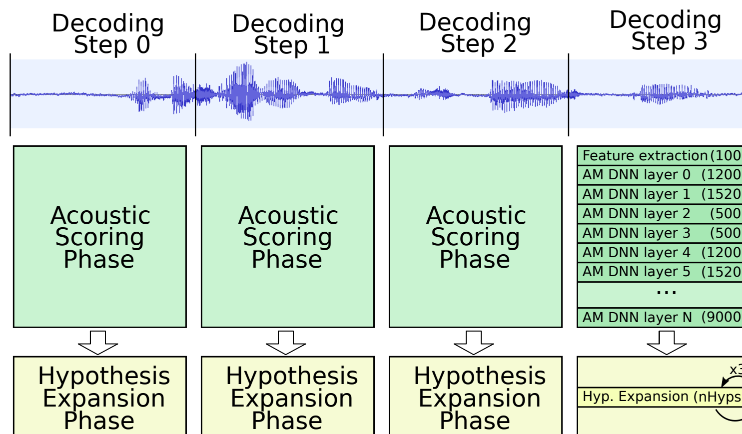


Figure 7.2: ASR process executed on ASRPU

pool of *Processing Elements (PE)* to execute the code of the kernels. The hypothesis unit sorts and prunes transcription hypotheses. It also keeps them in memory from one decoding step to the following.

7.1.1 Decoding on ASRPU

Figure 7.2 illustrates the overall process of decoding an utterance in ASRPU with an example ASR system. The decoding process in the accelerator is divided in *Decoding Steps*. Each step decodes a portion of the signal, extracting feature frames, computing acoustic scores and finally expanding the hypotheses left from the preceding decoding step. We divide each decoding steps in two phases: (1) The *Acoustic Scoring phase* and (2) the *Hypothesis Expansion phase*.

As previously mentioned, a set of kernels implement every component of the ASR system. The acoustic scoring phase consist of the sequential execution of most of these kernels (except for the last one). These kernels implement the feature extraction algorithm and the acoustic model. The

example of the figure shows a sequence of N+1 kernels executed within the acoustic scoring phase. These kernels are executed sequentially on the accelerator. However, they consist of parallel code. The execution of each kernel is carried on by the execution unit, which launches as many threads of the kernel code as required on the PEs. The number inside the parenthesis shows the number of threads required by each kernel. The first kernel implements feature extraction (which may include code for signal pre-processing) and requires 100 threads. Subsequent kernels implement each a layer of a DNN AM, each requiring a different number of threads. The last kernel requires 9000 threads, which is entirely dependant on the implementation. In this example, the last kernel implements a DNN layer with 9000 neurons. Each neuron computing the score for one of the 9000 phonetic units modelled by the acoustic model. because of how the kernel is written, each thread computes a single neuron.

After the acoustic scoring phase concludes, ASRPU switches to the hypothesis expansion phase. During the hypothesis expansion phase, the accelerator executes only one kernel, *Hypothesis expansion*. Each thread of this kernel is responsible of expanding a single hypothesis. The expansion of an hypothesis generally results in many output hypotheses, which are generated according to the specific decoding algorithm. Depending on the implementation, the acoustic scoring phase can generate one or more acoustic vectors. During hypothesis expansion, the accelerator executes the hypothesis expansion kernel once per acoustic vector. In the example system of the figure, the accelerator launches *nHyps* threads (determined in run-time) of the hypothesis expansion kernel. The self-referencing arrow indicates that the kernel is executed three times. This number will also depend on the implementation. For example, the feature extraction kernel may extract three frames on each decoding step, resulting in three repetitions of the hypothesis expansion kernel. Some DNNs, particularly convolutional DNNs, apply *sub-sampling* during acoustic scoring, meaning that they generate less acoustic vectors than feature frames. In this case, six feature frames will result in three acoustic vectors if the DNN AM apply a sub-sampling of two frames.

If ASRPU is integrated in an SoC that also contains a CPU, there may be an external process responsible of capturing the signal as it is produced. This process communicates with the accelerator, starting decoding steps after capturing enough values from the microphone.

7.1.2 Setup Thread

The ASR designer can include a special *setup program* along each of the acoustic scoring and hypothesis expansion kernels. That is, each kernel is complemented with a setup program. This setup program is executed to completion in a single thread before the associated kernel can start executing.

These setup programs provide the accelerator with greater flexibility. For example, the setup program for a specific kernel that implements a convolutional layer of a DNN can determine how many outputs can be computed from the available inputs and notify the hardware to launch the appropriate number of kernel threads so as to maximize data reuse. The setup program associated to the hypothesis expansion kernel can access the number of outputs generated by the acoustic scoring phase and notify it to the hardware so it executes the hypothesis expansion kernels as many times as necessary. In both cases, if the available inputs are not enough to compute even a single

output, the setup thread can notify to the accelerator to stop the decoding step. The following section provides mores details about this process.

These setup program can also be used to manage the input and output buffers of the kernels in shared memory. Each kernel will generally read inputs from an input buffer and store outputs in an output buffer (in shared memory). Before executing each kernel, the associated setup thread will first determine the number of outputs that can be generated from the inputs available in the input buffer. Then, it will remove from the input buffer those inputs that can not be further reused and reserve space in the output buffer for the new outputs. Finally, before finishing, it notifies the hardware the required number of kernel threads. After the setup thread finishes, the accelerator launches the required number of kernel threads.

Another advantage of the setup threads is that they allow to reuse code among different kernels. Generally, DNNs contain many convolutional and fully-connected layers. The ASR designer can write a single convolutional and a single fully-connected parametric routines. All the convolutional and fully-connected layers can be configured to execute the same kernels and the associated setup threads will set the appropriate parameters in shared memory before executing the kernel.

7.1.3 Execution Unit

The execution unit consists of a pool of PEs and an *ASR controller*. The ASR controller handles the overall decoding procedure. It first waits until the command decoder receives a new commit signal. At that moment, it starts a decoding step. First, the controller reads from the Configuration memory the address of the first setup program and configures a PE to execute it by setting its program counter. Once the setup thread finishes executing, it notifies to the ASR controller the required number of kernel threads. The ASR controller then starts dispatching kernel threads to idle PEs. Every time a PE becomes idle, it notifies the ASR controller, which reacts by dispatching a new thread to the PE, until there are no more threads to dispatch. When the last thread finishes, the ASR controller repeats the same procedure for the subsequent kernel.

As mentioned in section 7.1.2, if a setup thread returns a value of zero, the ASR controller stops the decoding step. This is meant to be used when a program is not ready to be launched, usually when there are not enough inputs to compute even a single output. For example, a convolutional layer with a window of ten frames will check during setup time (during the execution of the setup thread) how many inputs there are available, computing and returning an appropriate number of threads. If there are less than ten inputs, it will return zero notifying the ASR controller to stop the decoding step.

After all the programs in the Acoustic Scoring sequence have been executed, the decoder starts the hypothesis expansion phase. It first accesses the number of active hypotheses, provided by the hypothesis unit, and launches a thread for each active hypothesis. These threads will execute the code from the hypothesis expansion kernel. The setup thread of the hypothesis expansion kernel will determine how many outputs were generated by the acoustic scoring phase and notify the ASR controller to execute the hypothesis expansion kernel that number of times.

Figure 7.3 shows how the different threads are scheduled in the PE pool during acoustic scoring.

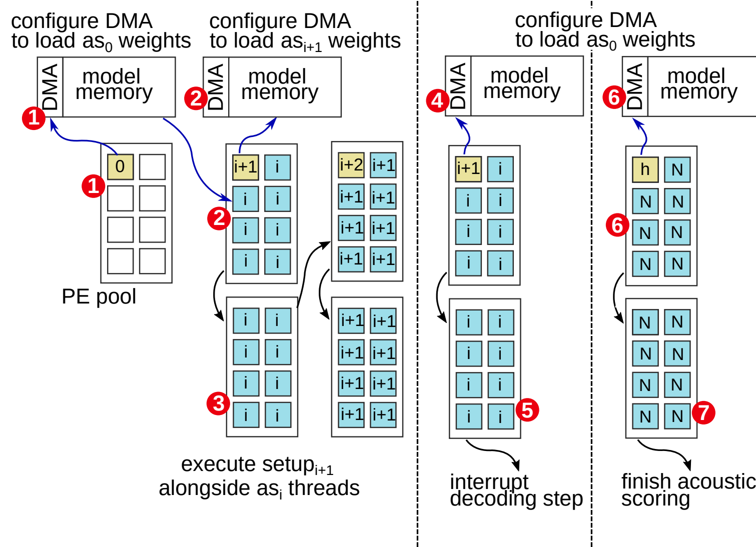


Figure 7.3: Threads in the PE pool

Each square represents a PE executing a setup thread (yellow) or a kernel thread (blue). ❶ First, the setup thread of kernel 0 is dispatched. It configures the DMA to load the model data for kernel 0 in model memory and waits for it to finish. ❷ The execution of the following kernels (as_i in the figure) starts by dispatching the setup thread for the next kernel (as_{i+1}) alongside the kernel threads of as_i. ❸ The ASR controller keeps dispatching as_i threads until the kernel is completely executed. If a setup thread determines that the corresponding thread cannot be launched ❹, it will notify the controller. Additionally, it can pre-fetch the model data for kernel 0 to skip step ❶ during the next decoding step. After the current kernel finishes ❺, the controller will interrupt the decoding step and wait for the next decoding command, which will start a new decoding step from ❶ or ❷, depending on whether the model data for kernel 0 is pre-loaded or not. ❻ The setup for the hypothesis expansion phase is launched alongside the threads for the last acoustic scoring kernel. Finally, when all the threads for the last acoustic scoring kernel finish ❼, the accelerator ends the acoustic scoring phase.

7.1.4 Processing Elements

The *Processing Element (PE)* pool contains a number of programmable and independent PEs (i.e. cores). Each PE, shown in figure 7.4, implements a general-purpose RISC-V ISA.

The ISA includes extensions for additional operations, such as a vector *Multiply and Accumulate (MAC)*. This operation receives three operands, the first operand is a 32-bit value that carries the accumulation between MAC operations. The other two operands are vectors of 8-bit values. These operands are multiplied element-wise and accumulated. The result is added to the first operand. It also includes vector multiplication and additions, along with especial function units to compute logarithms, exponential and cosine functions, usually required during feature extraction, the activation function in neural network layers and for the computation of the hypothesis score during hypothesis expansion. Each PE contains data and instruction caches. These are regular

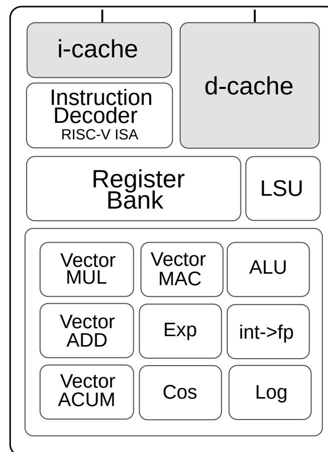


Figure 7.4: Processing Element (PE)

caches managed by the hardware. Each PE also contains a register bank with 2 sets of registers: 32-bit floating-point registers, which are used as operands for the FP ALU and the special function units, and vectors of 8-bit values used as operands for the vector operations. PEs are connected to the Hypothesis unit, the shared memory and the shared caches through a bus. Another bus connects all the PEs to the ASR controller. This bus is used by the ASR controller to configure the PEs and by the PEs to notify values to the ASR controller.

7.1.5 Hypothesis unit

The hypothesis unit contains a hypothesis memory and a controller. During any decoding step, the active hypothesis and the newly generated hypothesis reside inside the hypothesis memory. This unit is connected to the internal bus and accessed via a special memory address from the PEs. Hypothesis Expansion threads send hypotheses to the hypothesis controller. Each hypothesis is a data structure with some fields. These fields include a hash to identify the hypothesis, the hypothesis score, and others defined by the programmer. These can include a backlink, pointers to data structures (e.g. to a node in the decoding graph) or a token id, for example.

Hypothesis expansion threads access hypotheses from this unit and send back the newly generated hypotheses. The hypothesis unit sorts and prunes them according to their score field and the beam score. The score beam is configured beforehand via configuration commands.

7.1.6 Memory Hierarchy

Each PE contains data and an instruction cache. Outside the PE pool, there are shared instruction and data caches too. The global data performs two different functions. During acoustic scoring, this memory stores model weights that were pre-fetched beforehand. This maximizes data reuse and hides the latency to access external memory. During hypothesis expansion this scheme

would not be of much use. The graph structures used by the hypothesis expansion algorithms are generally in the order of hundreds of MB or even GB, much larger than what is reasonable to store in a low power accelerator. Additionally, the threads access the graph structures following a random pattern. Consecuently, during the hypothesis expansion phase, the data cache acts as a regular LRU cache to leverage locality in the access to the graph structures.

ASRPU also includes a scratchpad memory (the *Shared Memory*) that is be accessed from the threads executing in the PEs. This is were the kernel buffers and the kernel configuration parameters are stored, along with any other variables defined by the programmer.

7.1.7 Command Decoder

The command decoder is the interface between ASRPU and the rest of the units in the SoC. It provides a set of commands (table 7.1). These commands include some to configure the kernels and setup programs for the ASR phases: *ConfigureASR_AcousticScoring, ConfigureASR_HypExpansion* and commands to configure other parameters (*ConfigureBeamWidth*). These configuration commands must be used to configure the decoder before any decoding begins. In addition to those, the API contains commands for run-time operations. *DecodingStep* is to indicate the accelerator to decode a given signal. This signal is not decoded in isolation. Instead, it is appended to previously decoded signals, extending the current transcription hypotheses. Once the utterance is finished, *CleanDecoding* can be called. This command notifies the accelerator that the utterance is finished. In response, the accelerator prepares itself to decode a new utterance, cleaning the hypotheses memory and resetting the internal state.

7.2 Case Study

To illustrate the versatility and simplicity of our programming model for ASR, we present the implementation of one of the end-to-end systems from *wav2letter*. Features are 80-dim MFCCs computed from the pre-processed audio signal. The acoustic model is a TDS network, built from TDS blocks (figure 2.5). It is mostly composed of fully-connected and convolutional layers. The activation function for most layers is a ReLU, followed by a layer normalization. Hypotheses are extracted by traversing a lexicon tree that includes all the words in the vocabulary and a mechanism to handle out-of-vocabulary words. Additionally, an n-gram language model provides language model scores for the hypothesis.

In our implementation, the kernels that implement the acoustic scoring phase will first pre-process the signal and generate the MFCC frames. Then, they perform inference with the TDS network to obtain the acoustic scores from each of the computed frames. On each hypothesis expansion execution, all the hypotheses are expanded one node forward in the lexicon tree, covering each reachable node. Every reached node in the tree is a new hypothesis for the following hypothesis expansion execution. Every time a hypothesis reaches a node in the lexicon tree that represents a word, a link in the n-gram language model graph is traversed. The n-gram graph contains language model scores that are included, along with the acoustic scores, the word penalty and others, in the computation of the hypothesis score. Hypotheses are compared based on this score and those with

Table 7.1: Commands provided by the command decoder

Command	Parameters	Description
ConfigureASR_AcousticScoring	n_kernel setup_addr kernel_addr	Configure kernel n from the Acoustic Scoring phase. setup_addr and kernel_addr refer to the address in external memory pointing to the setup program and the kernel program, respectively. Should be called several times with incremental values of n to configure all the kernels that implement the acoustic scoring phase.
ConfigureASR_HypExpansion	kernel_addr	Configure the Hypothesis Expansion phase. kernel_addr is the address in external memory pointing to the hypothesis expansion kernel.
ConfigureBeamWidth	beam	Configure the *beam width* used by the hypothesis unit to prune hypotheses during hypothesis expansion.
CleanDecoding		Perform the neccesarry operations to start decoding a new utterance, such as removing the hypotheses from the hypothesis memory.
DecodingStep	signal_addr	Command the accelerator to start a decoding step. The accelerator will access the data located in signal_addr in the external memory and perform a decoding step.

a lower score are pruned away by the hypothesis unit. In addition to the reachable nodes in the lexicon tree, hypothesis expansion generates two more hypotheses as part of the CTC algorithm: the blank symbol and the repetition.

7.2.1 The Main Process

The main process residing in the CPU orchestrates the overall decoding of utterances. It does so by calling commands from the API of the accelerator. Before the decoding starts, the main process configures the accelerator, setting all the necessary parameters, including the addresses in external memory of the kernels that implement the ASR system.

During decoding, the main process collects reading from the microphone. This example ASR system performs streaming decoding, meaning that every few milliseconds, the main process calls the submittSignal command to perform a decoding step on a partial signal. If this was not the case, the main process would capture the signal until the end of the utterance is reached and then call a submittSignal on the entire signal.

7.2.2 Acoustic Scoring

The acoustic scoring phase executes the code that implements the feature extraction and the acoustic model. The acoustic scoring phase consists of a set of programs executed in sequence. In this case study, the first kernel performs signal pre-processing and extracts MFCC features frames from the input signal, whereas the rest implement each a layer of the TDS DNN.

Before executing the feature extraction kernel, its setup thread is launched to check the size of the input signal and determine how many output frames can be computed from the available input. Then, it reserves memory for the output, marks the inputs as consumed and notifies the controller about how many main threads must be launched. The kernel threads then process the inputs to generate feature frames. Each thread computes a single feature frame, which means that for each output frame to compute, a feature extraction thread will be launched.

The subsequent kernels in the acoustic scoring phase implement the TDS DNN. It is implemented in a sequence of 79 kernels: 18 CONV, 29 FC and 32 LayerNorms, each preceded by its corresponding setup thread. To avoid repeating very similar code, the programs for CONV, FC and LayerNorm are parameterized. The setup thread sets the parameters in shared memory to the values corresponding to the current layer, which are accessed by the layer threads.

Each setup thread checks the number of inputs available (those generated by the previous layer), reserves memory for the outputs and notifies the ASR controller to launch the required number of threads for the layer program. Each CONV and FC thread compute a single neuron of the layer.

7.2.3 Hypothesis Expansion

The hypothesis expansion kernel implements the CTC decoding algorithm with lexicon and language model. Each thread processes a single hypothesis. The algorithm first accesses the node in the lexicon graph associated with the hypothesis, then, it traverses all the output links to access reachable nodes, generating a new hypothesis for each of them. Each hypothesis also contain a link to the language model graph, pointing to the last n-gram in the hypothesis. If a newly reached node in the lexicon graph represents a word, the hypothesis expansion thread will acess the node in the language model graph associated with the hypothesis and expand it one node further following the link that represents the newly added word. The node contains a language model score that is added to the score of the hypothesis. In addition to the hypotheses generated by traversing the lexicon graph, the CTC algorithm implemented in the hypothesis expansion threads require the generation of two more hypotheses: the first one obtained by appending to the hypothesis the last phonetic unit in the hypothesis to account for repetitions, which produce valid CTC paths. The other hypothesis is obtained by appending the *blank unit*, which represents a frame that does not contain a phonetic unit.

7.3 Evaluation

This section provides estimations on the performance of ASRPU when running the ASR system described throughout the previous sections. The goal of this section is to provide proof of the capacity of the proposed design to enable real-time ASR on very low-power devices. To that purpose, we studied a possible implementation for the TDS-based system described in the previous section and estimated its performance on the accelerator, configured to enable real-time ASR with that system. Furthermore, we estimate the power consumption and area footprint of that specific configuration.

7.3.1 Methodology and Scope

To estimate performance, we count the number of instructions for each kernel. For example, a loop will usually consist of two instructions for the comparison and conditional jump, one instruction for the variable update and the instructions for the loop body, all multiplied by the average number of iterations. Additionally, one instruction is added for the variable initialization. We assume that every PE executes one instruction per cycle, so we divide the number of instructions by the clock frequency of the PEs to obtain execution time.

To estimate chip area, we rely on several tools. Cacti for the memories, McPat for the PEs and the PE bus and Design compiler (using the *Saed32hvt cell library*, which provides cell models at 32nm technology node) for the special function units.

Peak power is estimated by adding together the leakage power and peak dynamic power for the logic units as obtained from the Power Compiler. The case of memories is slightly different. Cacti reports leakage power and access energy. In this case, we assume as peak power the scenario where all the ports are accessed once per cycle. Adding the energy consumed for those accesses, divided by the clock period gives the dynamic power, which we add, along with the leakage power given by Cacti, to the power consumed by the logic.

This estimation, albeit not exhaustive, should provide a good approximation of the potential of the accelerator proposed in this work.

7.3.2 Accelerator Configuration

Table 7.2 contains the details of the accelerator. This configuration was chosen to allow real-time ASR with the ASR system described in previous sections. Particularly, the number of PEs and the size of the memories was chosen to match the performance requirements. We include 8 PEs, each loaded with an 8-dim MAC unit, which allows us to exploit plenty of parallelism. The implemented algorithm stores about $275KB$ of intermediate data in between decoding steps. It stores inputs for the convolutional layers. Due to the shifting input window used in convolutions, inputs are reused in several consecutive executions. We include $512KB$ of shared memory to store these inputs and other temporal outputs that may be necessary to store if the decoding step is interrupted due to insufficient inputs for one of the kernels.

Table 7.2: Configuration parameters of the accelerator

ASR Unit

Frequency	500 MHz
Hypothesis Memory	24 KB
I-Cache	64 KB
Shared Memory	512 KB
Model Memory / D-Cache	1 MB
Num. PEs	8

PE

PE i-Cache	4 KB
PE d-Cache	24 KB
MAC. vector size	8

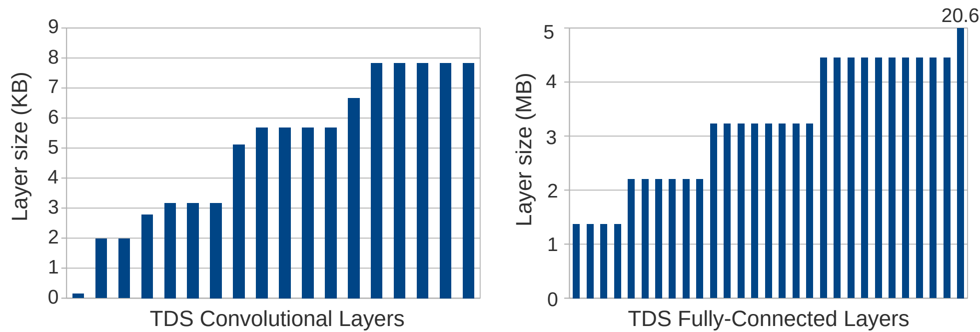
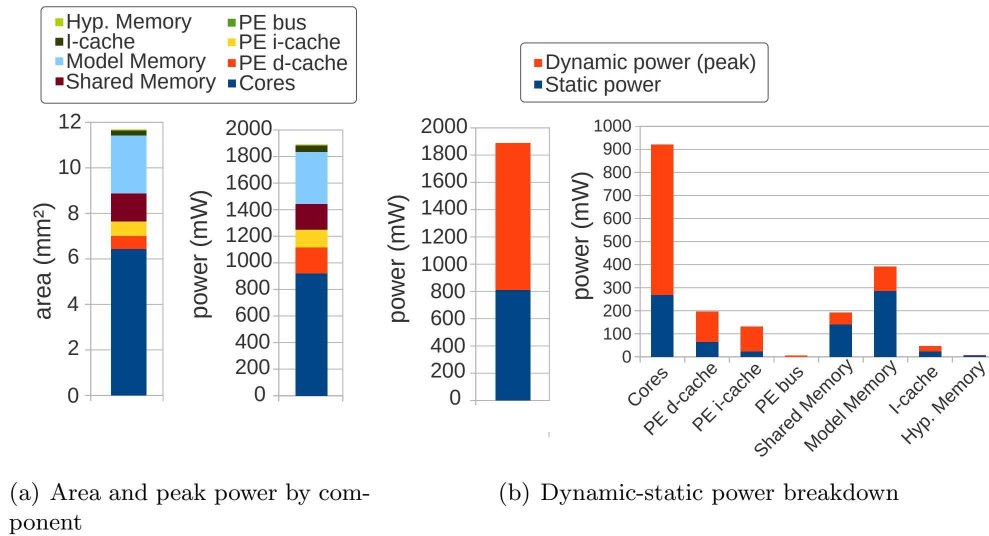


Figure 7.5: Size (KB) of each layer of the TDS DNN included in the ASR system. The left plot shows the convolutional layers whereas the right plot shows the fully-connected layers

We include $1MB$ of model memory. During acoustic scoring, this memory is used for caching the DNN parameters and other model data. The size of the TDS network layers vary significantly (figure 7.5). Convolutional layers fit in a few KB whereas most fully connected layers range in the MB. We solve this by trivially partitioning FC layers into several kernels, each less than $1MB$. Given that each thread in our implementation of the FC kernels computes a neuron, we partition the layers in various kernels, each computing some of the neurons. For example, each of the first FC layers consists of 1200 neurons with 1200 inputs each, which results in $1.4MB$ of model data. We divide each of these layers into 2 kernels, each computing 600 neurons ($700MB$).

7.3.3 Area and Power

Figure 7.6 shows an estimation of the area and peak power of ASRPU, broken down by component. At a 32 nm technology node, the total area is $11.68mm^2$, 65% of which is dedicated to the execution unit (PEs, PE d-cache, PE i-cache and PE bus), 32% is dedicated to the shared and model memories. The hypothesis unit accounts for less than 1 %. Regarding power, the accelerator consumes slightly more than 1.8 W assuming peak power. That is, if every PE is in execution and



(a) Area and peak power by component

(b) Dynamic-static power breakdown

Figure 7.6: The left bar plots show the component-level breakdown of area and peak power of ASRPU. The right plots show the distribution of static and dynamic power

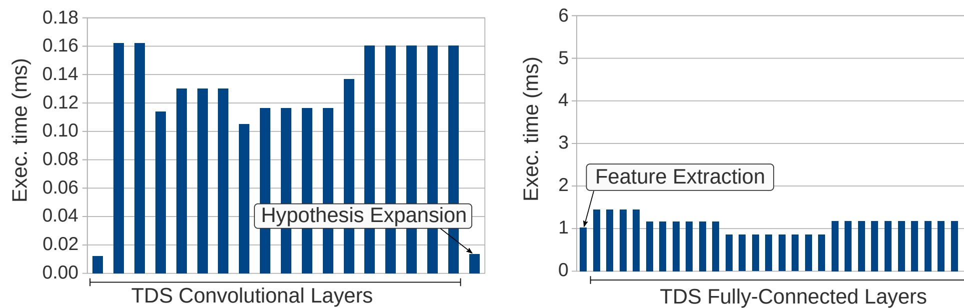


Figure 7.7: Execution time for the TDS ASR system running in ASRPU

every memory is accessed. Around 800 mW come from static power, mostly from the PE cores and the shared and model memories, whereas the rest comes from dynamic power, mainly from the PE cores.

7.3.4 Performance

Each decoding step in our implementation decodes 80ms of audio. According to our estimations, ASRPU takes about 40ms to perform a decoding step. In other words, the accelerator executes the ASR system in 2x real-time. Figure 7.7 shows the execution time of the ASR system kernels, including the feature extraction and the hypothesis expansion kernels. The left plot shows the execution time for the kernels that implement the convolutional layers and the hypothesis expansion, whereas the right plot shows the execution time taken by fully-connected layers and the

feature expansion during the execution of a decoding step. These estimations assume no network contention. We also assume that the model data is pre-fetched in model memory.

8
Conclusions and Future Work

This chapter presents a summary of our contributions and the results presented in this thesis. Furthermore, we include a discussion on the relevance of the computer architecture discipline to enable widespread adoption of applications based on artificial intelligence and machine learning and suggest some future directions for research in accelerated ASR. The first section is the summary of the results presented in this thesis, the second section is a summary of our contributions and the last section contains the discussion of future work.

8.1 Conclusions

ASR is a technology with broad implications that will be vital in the transition towards *ambience computing*. Now that decoding accuracy is so high that mainstream applications are possible, innovations that improve ASR energy efficiency and performance in low-power hardware are key to accelerating the adoption of the technology.

The objective of this work is to propose hardware innovations that improve the performance and energy efficiency of the ASR system. This work is framed within the more ambitious objective of enabling efficient real-time ASR on very low-power edge devices. To that aim, we first studied the range of available ASR systems and the literature on optimizations for ASR, including hardware accelerators, techniques to optimize the ASR models, such as quantization and pruning, and techniques to leverage run-time properties, such as those to predict ineffectual computations, and then proposed and evaluated different optimization techniques based on hardware acceleration and the specific properties of ASR systems to further improve performance and energy consumption.

In our first work (chapter 4), we study a Hybrid DNN-HMM ASR system included in the Kaldi framework. This system implements a Hybrid DNN-HMM ASR architecture composed of a TDNN

acoustic model, an HCLG decoding graph and a 4-gram LM for lattice rescoring. This system provides highly-accurate transcriptions with a vocabulary of around 200k words. We evaluate the execution of this ASR system on a platform that contains 8 GB of LPDDR4 memory and a quad-core ARM CPU and determine that less than 1% of the utterances in the Librispeech benchmark run faster than real-time. Including a mobile Nvidia Maxwell GPU with 256 cores enables real-time speech recognition for many of the utterances. However, not only that is at the expense of high power consumption, but also, this solution results in higher performance variability, with some utterances taking up to almost 10 times real-time to decode. We propose a heterogeneous platform that, instead of the GPU, includes an accelerator for DNN inference and a Viterbi accelerator. The heterogeneous platform performs, on average, 4.5x faster than the solution that includes the mobile GPU, while consuming 4.3x less energy per decoded frame. We estimate that the average power dissipation is slightly under 1 W. Additionally, our estimations show that the accelerations incur a chip area overhead of just 3.64 mm^2 using 28 nm technology nodes.

During our second work (chapter 5), we determined that the major bottleneck of the previous heterogeneous platform is the execution of the TDNN acoustic model and propose to leverage an interesting run-time property of ASR systems. The ASR system is less prone to make transcription mistakes derived from very-low arithmetic precision when the system is considering a low relative number of hypotheses, as opposed to when it considers a high number of them. We propose to measure the run-time variation of the number of active hypotheses and switch the TDNN acoustic model to an aggressively quantized version. We tested this proposal with the TDNN system, quantized to 8 bits and 4 bits, running on the heterogeneous platform that we modified to measure the number of active hypotheses and to efficiently accommodate support for arithmetic operations in 8 bits and 4 bits. Using this technique reduces the execution time of the TDNN inference in 25.8% and the energy consumption in 25.6%, which is translated to 16.9% less energy and 19.5% less execution time for the whole ASR system. This comes at the cost of less than 1% absolute WER loss and overhead of 0.12 mm^2 in chip area and about 0.17 W in average power.

For our third work (chapter 6), we shifted the focus towards end-to-end ASR systems, which have been gaining a lot of traction during the last few years and recently reached state-of-the-art accuracy in common benchmarks, such as Librispeech. The ASR system we studied for this work consists of a TDS network trained to perform CTC decoding. The outputs of the network are composed into a set of alternative hypotheses, that have scored based on the acoustic scores computed by the TDS network and LM scores obtained from a 4-gram language model. These DNNs make extensive use of ReLu activation layers, which, according to our observations, result in about 80 % of the outputs of the neurons being zero. We propose a very low-cost prediction scheme (Mixture-of-Rookies) to detect which neurons will produce a zero, and thus skip the computation of the neuron altogether. The key characteristic of our predictor is the combination of two prediction components, integrated into a DNN accelerator, to leverage both neuron self-correlation and neuron spatial correlation. Applying this scheme provides a 1.2x speedup during the TDS inferences while saving about 17 % of the energy consumption compared to a baseline consisting of the same accelerator without the prediction scheme. The additional hardware represents an overhead of 5.3 % in the chip area. Furthermore, we tested this technique on other DNN applications and obtained similar benefits while keeping the degradation of accuracy below 1% absolute loss of the relevant metric.

Our fourth work (chapter 7) proposes a programmable low-power accelerator for ASR (ASRPU) to tackle all the challenges of deploying state-of-the-art ASR on edge devices. This accelerator contains a pool of programmable cores, a memory hierarchy especially suited for ASR and several controllers to automate common characteristics of ASR systems and simplify their programming. We implement the TDS end-to-end system in ASRPU to demonstrate the flexibility of the programming model and estimate that it can run the TDS system at 2x real-time in streaming mode (batch-1) when configured in a very low power setup. According to our estimations, that setup requires 11.7 mm^2 of chip area and consumes about 1.8 W of power assuming peak performance.

8.2 Contributions

In this thesis, several techniques to improve the energy efficiency and performance of ASR are proposed. This work aims to enable highly accurate ASR in very low-power, battery-dependant edge devices. As a result of this thesis, we propose some hardware accelerators for ASR and optimizations techniques based on run-time characteristics of the ASR algorithms. The following paragraphs summarize the main contributions of this thesis.

Our first contribution is to characterize a state-of-the-art hybrid DNN-HMM ASR system running in a low-power platform. According to our experiments, the vast majority of utterances were not running in real-time, and even when the neural network inferences are offloaded to a mobile GPU, many utterances still run slower than in real-time. This characterization led to the design of a heterogeneous hardware platform that includes, along with a mobile CPU and a DRAM central memory, two hardware accelerators. In this proposal, we offload the most compute-demanding components of the ASR to the accelerators, which results in a 4.5x speedup and 4.3x less energy consumption. These results were compiled into a paper that was presented at the HiPEAC conference and published in the *ACM Transactions on Architecture and Code Optimization (TACO)* journal.

PINTO, Dennis; ARNAU, Jose-María; GONZÁLEZ, Antonio. Design and Evaluation of an Ultra Low-power Human-quality Speech Recognition System. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2020, vol. 17, no 4, p. 1-19.

Our next contribution was to change the arithmetic precision used during the execution of the DNN inference by following a novel mechanism that allows us to know whether the ASR system is operating with high or low confidence. When it operates with high confidence, it can handle more arithmetic noise, and thus, reducing the arithmetic precision has a lower impact on transcription accuracy. This approach results in 16.9% less energy and 19.5% less execution time during the execution of the whole ASR system. This work is has been submitted for publication and is currently under review.

PINTO, Dennis; ARNAU, Jose-María; GONZÁLEZ, Antonio. Exploiting Beam Search Confidence for Energy-Efficient Speech Recognition. *Under Review.*

Following that work, we observed that many DNN models, including those used for ASR, contain many ReLu activation functions, which generate plenty of zeros in run-time. This property

creates a huge opportunity to save computations that can be leveraged by predicting, in run-time, which neurons will output zero. We propose Mixture-of-Rookies, a prediction approach that can leverage self-correlation and spatial correlation among neurons at the same time with a very low overhead. This technique results in 1.2x faster decoding and 17 % less energy consumption. This work has been submitted for publication and is currently under review.

PINTO, Dennis; ARNAU, Jose-María; GONZÁLEZ, Antonio. Mixture-of-Rookies: Saving DNN Computations by Predicting ReLU Outputs. *Under Review*.

Finally, our last contribution is a hardware accelerator for ASR called ASRPU. This accelerator is programmable and contains a memory hierarchy and some modules specific to ASR. This allows for the efficient implementation of a wide range of ASR systems. Our estimations show that a state-of-the-art ASR system can be easily implemented for ASRPU and executed in real-time with power consumption below 1.8 W. This work has been submitted for publication and is currently under review.

PINTO, Dennis; ARNAU, Jose-María; GONZÁLEZ, Antonio. ASRPU: A Programmable Accelerator for Low-Power Automatic Speech Recognition. *Under Review*.

8.3 Future Work

The work proposed in this thesis can be extended by proposing optimizations for sequence-to-sequence models. We focus our study on a hybrid DNN-HMM model with a feed-forward acoustic model DNN (Section 4.1) and a CTC-based end-to-end system with a convolutional acoustic model DNN (Section 6.1). However, Sequence-to-sequence end-to-end models can also reach state-of-the-art accuracy and have become very popular recently. In these systems, the acoustic model DNN is divided into two blocks: the *encoder* and the *decoder*. The encoder processes the signal frames, transforming them into internal representations. In each decoding step, the decoder receives these *encoded* frames along with the symbol produced by the previous decoding step and generates a new symbol. These symbols, which may be characters, words or any other symbol, are concatenated to form a transcription. This process is repeated until the decoder produces a special *end-of-sentence* symbol. Novel optimizations can leverage properties specific to these systems to obtain gains.

Another class of ASR systems that has gained a lot of traction during the last few years are those based on Transformer networks [108, 33, 52]. These DNNs extend the idea of the *Attention Mechanism* introduced in *Listen, Attend and Spell (LAS)* [18] network to design a building block that can be used to build a DNNs, similar to the ResNet of TDS networks. This building block is called *Self-attention block* and the neural networks built from them are called *Transformer networks*. These networks have been proved very useful for ASR and their specific structure is very different from other DNNs. We think that there is still a lot of potentials to optimize ASR systems based on transformer networks. The attention scores are generally very sparse, exposing the huge potential for optimizations [110, 42]. Furthermore, studying the patterns of attention that arise during decoding may lead to deeper intuitions that can be leveraged to obtain further gains [47]. Therefore, we think that a deeper understanding of these patterns may help in determining when and how to optimize.

We believe that heterogeneous platforms that contain several hardware accelerators represent the future of computing. Offloading compute-intensive or memory-intensive workloads into specialized accelerators is probably going to be key in the deployment of AI-based applications in the edge. However, as discussed in chapter 7, relying on very specialized accelerators is a limited approach. Further research in programmable architectures for restricted workloads may prove extremely beneficial. In this sense, our ASRPU accelerator can be extended to support attention-based and sequence-to-sequence networks. Furthermore, ASR optimizations and more specialized hardware structures can be introduced in the accelerator. The ISA of the PEs can be specialized, removing unnecessary instructions and including specific instructions that make sense in the context of ASR. For example, an instruction that broadcast data to all the data caches of the PEs may save memory access and free bandwidth consumption.

A key property of ASR is that it is a dynamic process. Every audio frame is different. Some are more difficult to decode than others. Furthermore, not every word in the transcription is equally important for the meaning of the sentence. All these characteristics make ASR systems prone to run-time optimizations. Throughout this work, we studied run-time properties of ASR algorithms to save energy and reduce latency (Chapters 5 and 6). We think that there is still a lot of potential for optimizations based on this kind of property. Similar to our solution in chapter 5, aggressive optimizations strategically applied at moments when the accuracy is unlikely to be affected may provide huge gains in future accelerated ASR. Therefore, we think that studying these properties is one of the most interesting research directions to improve ASR on the edge.

Recent innovations in 3D memories, near-data computing and in-memory computing will likely be very beneficial for ASR. ASR is trending towards bigger and more complex DNN acoustic models. Streaming or batch-1 ASR is generally memory intensive. The resulting bottlenecks in memory access prevent the usage of the extensive sources of parallelism present in these algorithms. Further research in these innovative memory architectures and their application to ASR may unlock further gains.

Bibliography

[1] Alexa. https://en.wikipedia.org/wiki/Amazon_Alexa. [Online; accessed 29-Oct-2021].

[2] Cortana. https://www.microsoft.com/en-us/cortana. [Online; accessed 29-Oct-2021].

[3] Cuda. https://docs.nvidia.com/cuda. [Online; accessed 29-Oct-2021].

[4] Nvdla. http://nvdla.org/. [Online; accessed 29-Oct-2021].

[5] Opencl. https://opencl.org/. [Online; accessed 29-Oct-2021].

[6] siri. https://en.wikipedia.org/wiki/Siri. [Online; accessed 29-Oct-2021].

[7] Speech recognition on librispeech test-clean. https://paperswithcode.com/sota/speech-recognition-on-librispeech-test-clean. [Online; accessed 29-Oct-2021].

[8] Speech and voice recognition market size, share & industry analysis, by component (solution, services), by technology (voice recognition, speech recognition), by deployment (on-premises, cloud), by end-user (healthcare, it and telecommunications, automotive, bfsi, government, legal, retail, travel and hospitality and others) and regional forecast, 2019 – 2026, 2019.

[9] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[10] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K Gupta, and Hadi Esmaeilzadeh. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 662–673. IEEE, 2018.

[11] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. *ACM SIGARCH Computer Architecture News*, 44(3):1–13, 2016.

[12] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182. PMLR, 2016.

[13] Alexander G Anderson and Cory P Berg. The high-dimensional geometry of binary neural networks. *arXiv preprint arXiv:1705.07199*, 2017.

[14] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[15] Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations. *arXiv preprint arXiv:2006.11477*, 2020.

[16] Herve A Bourlard and Nelson Morgan. *Connectionist speech recognition: a hybrid approach*, volume 247. Springer Science & Business Media, 2012.

[17] Shijie Cao, Lingxiao Ma, Wencong Xiao, Chen Zhang, Yunxin Liu, Lintao Zhang, Lanshun Nie, and Zhi Yang. Seernet: Predicting convolutional neural network feature-map sparsity through low-bit quantization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11216–11225, 2019.

[18] William Chan, Navdeep Jaitly, Quoc Le, and Oriol Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4960–4964. IEEE, 2016.

[19] William Chan, Daniel Park, Chris Lee, Yu Zhang, Quoc Le, and Mohammad Norouzi. Speechstew: Simply mix all available speech recognition data to train one large neural network. *arXiv preprint arXiv:2104.02133*, 2021.

[20] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.

[21] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

[22] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[23] Chung-Cheng Chiu, Tara N Sainath, Yonghui Wu, Rohit Prabhavalkar, Patrick Nguyen, Zhifeng Chen, Anjuli Kannan, Ron J Weiss, Kanishka Rao, Ekaterina Gonina, et al. State-of-the-art speech recognition with sequence-to-sequence models. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4774–4778. IEEE, 2018.

[24] Anthony Chun, Jenny X Chang, Zhen Fang, Ravishankar Iyer, and Michael Deisher. Isis: An accelerator for sphinx speech recognition. In *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, pages 58–61. IEEE, 2011.

[25] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[26] Xiaoliang Dai, Hongxu Yin, and Niraj K Jha. Grow and prune compact, fast, and accurate lstms. *IEEE Transactions on Computers*, 69(3):441–452, 2019.

[27] Namrata Dave. Feature extraction methods lpc, plp and mfcc in speech recognition. *International journal for advance research in engineering and technology*, 1(6):1–4, 2013.

[28] Chunhua Deng, Siyu Liao, Yi Xie, Keshab K Parhi, Xuehai Qian, and Bo Yuan. Permdnn: Efficient compressed dnn architecture with permuted diagonal matrices. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 189–202. IEEE, 2018.

[29] Xiaohan Ding, Guiguang Ding, Yuchen Guo, and Jungong Han. Centripetal sgd for pruning very deep convolutional networks with complicated structure. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4943–4953, 2019.

[30] Xuanyi Dong, Junshi Huang, Yi Yang, and Shuicheng Yan. More is less: A more complicated network with less inference complexity. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5840–5848, 2017.

[31] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 92–104, 2015.

[32] Michael Figurnov, Aijan Ibraimova, Dmitry Vetrov, and Pushmeet Kohli. Perforatedcnns: Acceleration through elimination of redundant convolutions. *arXiv preprint arXiv:1504.08362*, 2015.

[33] Dominique Fohr and Irina Illina. Bert-based semantic model for rescoring n-best speech recognition list. In *INTERSPEECH 2021*, 2021.

[34] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

[35] John S Garofolo, Lori F Lamel, William M Fisher, Jonathan G Fiscus, and David S Pallett. Darpa timit acoustic-phonetic continous speech corpus cd-rom. nist speech disc 1-1.1. *NASA STI/Recon technical report n*, 93:27403, 1993.

[36] John J Godfrey, Edward C Holliman, and Jane McDaniel. Switchboard: Telephone speech corpus for research and development. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, volume 1, pages 517–520. IEEE Computer Society, 1992.

[37] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376, 2006.

[38] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. Hybrid speech recognition with deep bidirectional lstm. In *2013 IEEE workshop on automatic speech recognition and understanding*, pages 273–278. IEEE, 2013.

[39] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, et al. Conformer: Convolution-augmented transformer for speech recognition. *arXiv preprint arXiv:2005.08100*, 2020.

[40] Kartiki Gupta and Divya Gupta. An analysis on lpc, rasta and mfcc techniques in automatic speech recognition system. In *2016 6th international conference-cloud system and big data engineering (confluence)*, pages 493–497. IEEE, 2016.

[41] Udit Gupta, Brandon Reagen, Lillian Pentecost, Marco Donato, Thierry Tambe, Alexander M Rush, Gu-Yeon Wei, and David Brooks. Masr: A modular accelerator for sparse rnns. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–14. IEEE, 2019.

[42] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, et al. Aˆ 3: Accelerating attention mechanisms in neural networks with approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 328–341. IEEE, 2020.

[43] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84, 2017.

[44] Wei Han, Cheong-Fat Chan, Chiu-Sing Choy, and Kong-Pang Pun. An efficient mfcc extraction method in speech recognition. In *2006 IEEE international symposium on circuits and systems*, pages 4–pp. IEEE, 2006.

[45] Wei Han, Zhengdong Zhang, Yu Zhang, Jiahui Yu, Chung-Cheng Chiu, James Qin, Anmol Gulati, Ruoming Pang, and Yonghui Wu. Contextnet: Improving convolutional neural networks for automatic speech recognition with global context. *arXiv preprint arXiv:2005.03191*, 2020.

[46] Awni Hannun, Ann Lee, Qiantong Xu, and Ronan Collobert. Sequence-to-sequence speech recognition with time-depth separable convolutions. *arXiv preprint arXiv:1904.02619*, 2019.

[47] Yaru Hao, Li Dong, Furu Wei, and Ke Xu. Self-attention attribution: Interpreting information interactions inside transformer. *arXiv preprint arXiv:2004.11207*, 2020.

[48] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[49] Yanzhang He, Tara N Sainath, Rohit Prabhavalkar, Ian McGraw, Raziel Alvarez, Ding Zhao, David Rybach, Anjuli Kannan, Yonghui Wu, Ruoming Pang, et al. Streaming end-to-end speech recognition for mobile devices. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6381–6385. IEEE, 2019.

[50] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[51] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[52] Irina Illina and Dominique Fohr. Dnn-based semantic rescoring models for speech recognition. In *TSD 2021-24th International Conference on Text, Speech and Dialogue*, 2021.

[53] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[54] Chadawan Ittichaichareon, Siwat Suksri, and Thaweesak Yingthawornsuk. Speech recognition using mfcc. In *International conference on computer graphics, simulation and modeling*, pages 135–138, 2012.

[55] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[56] Patrick Judd, Alberto Delmas, Sayeh Sharify, and Andreas Moshovos. Cnvlutin2: Ineffectual-activation-and-weight-free deep neural network computing. *arXiv preprint arXiv:1705.00125*, 2017.

[57] Jacob Kahn, Morgane Rivière, Weiyi Zheng, Evgeny Kharitonov, Qiantong Xu, Pierre-Emmanuel Mazaré, Julien Karadayi, Vitaliy Liptchinsky, Ronan Collobert, Christian Fuegen, et al. Libri-light: A benchmark for asr with limited or no supervision. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7669–7673. IEEE, 2020.

[58] Jodi Kearns. Librivox: Free public domain audiobooks. *Reference Reviews*, 2014.

[59] Manish P Kesarkar and P Rao. Feature extraction for speech recognition. *Electronic Systems, EE. Dept., IIT Bombay*, 2003.

[60] Cheolhwan Kim, Dongyeob Shin, Bohun Kim, and Jongsun Park. Mosaic-cnn: A combined two-step zero prediction approach to trade off accuracy and computation energy in convolutional neural networks. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 8(4):770–781, 2018.

[61] Idan Kligvasser, Tamar Rott Shaham, and Tomer Michaeli. xunit: Learning a spatial activation function for efficient image restoration. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2433–2442, 2018.

[62] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[63] Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. *arXiv preprint arXiv:1804.10959*, 2018.

[64] Anuj Kumar, Anuj Tewari, Seth Horrigan, Matthew Kam, Florian Metze, and John Canny. Rethinking speech recognition on mobile devices. 2011.

[65] Duc Le, Xiaohui Zhang, Weiyi Zheng, Christian Fügen, Geoffrey Zweig, and Michael L Seltzer. From senones to chenones: Tied context-dependent graphemes for hybrid speech recognition. In *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 457–464. IEEE, 2019.

[66] Jason Li, Vitaly Lavrukhin, Boris Ginsburg, Ryan Leary, Oleksii Kuchaiev, Jonathan M Cohen, Huyen Nguyen, and Ravi Teja Gadde. Jasper: An end-to-end convolutional neural acoustic model. *arXiv preprint arXiv:1904.03288*, 2019.

[67] Jinyu Li, Yu Wu, Yashesh Gaur, Chengyi Wang, Rui Zhao, and Shujie Liu. On the comparison of popular end-to-end models for large scale speech recognition. *arXiv preprint arXiv:2005.14327*, 2020.

[68] Ke Li, Daniel Povey, and Sanjeev Khudanpur. A parallelizable lattice rescoring strategy with neural language models. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6518–6522. IEEE, 2021.

[69] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009.

[70] Edward C Lin, Kai Yu, Rob A Rutenbar, and Tsuhan Chen. Moving speech recognition from software to silicon: the in silico vox project. In *Ninth International Conference on Spoken Language Processing*, 2006.

[71] Yingyan Lin, Charbel Sakr, Yongjune Kim, and Naresh Shanbhag. Predictivenet: An energy-efficient convolutional neural network via zero prediction. In *2017 IEEE international symposium on circuits and systems (ISCAS)*, pages 1–4. IEEE, 2017.

[72] Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. Autocompress: An automatic dnn structured pruning framework for ultra-high compression rates. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 4876–4883, 2020.

[73] Tao Luo, Shaoli Liu, Ling Li, Yuqing Wang, Shijin Zhang, Tianshi Chen, Zhiwei Xu, Olivier Temam, and Yunji Chen. Dadiannao: A neural network supercomputer. *IEEE Transactions on Computers*, 66(1):73–88, 2016.

[74] Xiaolong Ma, Fu-Ming Guo, Wei Niu, Xue Lin, Jian Tang, Kaisheng Ma, Bin Ren, and Yanzhi Wang. Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5117–5124, 2020.

[75] Mostafa Mahmoud, Kevin Siu, and Andreas Moshovos. Diffy: A déjà vu-free differential deep neural network accelerator. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 134–147. IEEE, 2018.

[76] Inc Micron Technology. Tn-53-01: Lpddr4 system power calculator. urlhttps://www.micron.com/support/tools-and-utilities/power-calc, 2016.

[77] Kazuo Miura, Hiroki Noguchi, Hiroshi Kawaguchi, and Masahiko Yoshimoto. A low memory bandwidth gaussian mixture model (gmm) processor for 20,000-word real-time speech recognition fpga system. In *2008 International Conference on Field-Programmable Technology*, pages 341–344. IEEE, 2008.

[78] Mehryar Mohri, Fernando Pereira, and Michael Riley. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88, 2002.

[79] Mehryar Mohri, Fernando Pereira, and Michael Riley. Speech recognition with weighted finite-state transducers. In *Springer Handbook of Speech Processing*, pages 559–584. Springer, 2008.

[80] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27:28, 2009.

[81] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an asr corpus based on public domain audio books. In *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5206–5210. IEEE, 2015.

[82] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

[83] Douglas B Paul and Janet Baker. The design for the wall street journal-based csr corpus. In *Speech and Natural Language: Proceedings of a Workshop Held at Harriman, New York, February 23-26, 1992*, 1992.

[84] Vijayaditya Peddinti, Daniel Povey, and Sanjeev Khudanpur. A time delay neural network architecture for efficient modeling of long temporal contexts. In *Sixteenth annual conference of the international speech communication association*, 2015.

[85] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely. The kaldi speech recognition toolkit. In *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society, December 2011.

[86] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely. The kaldi speech recognition toolkit. In *IEEE 2011 workshop on automatic speech recognition and understanding*, number CONF. IEEE Signal Processing Society, 2011.

[87] Vineel Pratap, Awni Hannun, Qiantong Xu, Jeff Cai, Jacob Kahn, Gabriel Synnaeve, Vitaliy Liptchinsky, and Ronan Collobert. wav2letter++: The fastest open-source speech recognition system. *CoRR*, abs/1812.07625, 2018.

[88] Vineel Pratap, Awni Hannun, Qiantong Xu, Jeff Cai, Jacob Kahn, Gabriel Synnaeve, Vitaliy Liptchinsky, and Ronan Collobert. Wav2letter++: A fast open-source speech recognition

system. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6460–6464. IEEE, 2019.

[89] Vineel Pratap, Qiantong Xu, Jacob Kahn, Gilad Avidov, Tatiana Likhomanenko, Awni Hannun, Vitaliy Liptchinsky, Gabriel Synnaeve, and Ronan Collobert. Scaling up online speech recognition using convnets. *arXiv preprint arXiv:2001.09727*, 2020.

[90] Michael Price et al. *Energy-scalable speech recognition circuits.* PhD thesis, Massachusetts Institute of Technology, 2016.

[91] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70. IEEE, 2020.

[92] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[93] Haşim Sak, Murat Saraclar, and Tunga Güngör. On-the-fly lattice rescoring for real-time automatic speech recognition. In *Eleventh annual conference of the international speech communication association*, 2010.

[94] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic cnn accelerator simulator. *arXiv preprint arXiv:1811.02883*, 2018.

[95] Johan Schalkwyk, Doug Beeferman, Françoise Beaufays, Bill Byrne, Ciprian Chelba, Mike Cohen, Maryam Kamvar, and Brian Strope. "your word is my command": Google search by voice: A case study. In *Advances in speech recognition*, pages 61–90. Springer, 2010.

[96] Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152. IEEE, 2012.

[97] Gil Shomron, Ron Banner, Moran Shkolnik, and Uri Weiser. Thanks for nothing: Predicting zero-valued activations with lightweight convolutional neural networks. In *European Conference on Computer Vision*, pages 234–250. Springer, 2020.

[98] Gil Shomron and Uri Weiser. Spatial correlation and value prediction in convolutional neural networks. *IEEE Computer Architecture Letters*, 18(1):10–13, 2018.

[99] Urmila Shrawankar and Vilas M Thakare. Techniques for feature extraction in speech recognition system: A comparative study. *arXiv preprint arXiv:1305.1145*, 2013.

[100] Franyell Silfa, Gem Dot, Jose-Maria Arnau, and Antonio Gonzàlez. Neuron-level fuzzy memoization in rnns. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 782–793, 2019.

[101] James E Smith. Decoupled access/execute computer architectures. In *ACM SIGARCH Computer Architecture News*, volume 10, pages 112–119. IEEE Computer Society Press, 1982.

[102] Mingcong Song, Jiechen Zhao, Yang Hu, Jiaqi Zhang, and Tao Li. Prediction based execution on deep neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 752–763. IEEE, 2018.

[103] Savitha Srinivasan and Eric Brown. Is speech recognition becoming mainstream? *Computer*, 35(04):38–41, 2002.

[104] Stanley Smith Stevens, John Volkmann, and Edwin Broomell Newman. A scale for the measurement of the psychological magnitude pitch. *The journal of the acoustical society of america*, 8(3):185–190, 1937.

[105] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[106] Gabriel Synnaeve, Qiantong Xu, Jacob Kahn, Edouard Grave, Tatiana Likhomanenko, Vineel Pratap, Anuroop Sriram, Vitaliy Liptchinsky, and Ronan Collobert. End-to-end asr: from supervised to semi-supervised learning with modern architectures. *arXiv preprint arXiv:1911.08460*, 2019.

[107] Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, and Antonio Gonzalez. An ultra low-power hardware accelerator for acoustic scoring in speech recognition. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 41–52. IEEE, 2017.

[108] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[109] Dong Wang, Xiaodong Wang, and Shaohe Lv. An overview of end-to-end automatic speech recognition. *Symmetry*, 11(8):1018, 2019.

[110] Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 97–110. IEEE, 2021.

[111] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29:2074–2082, 2016.

[112] Paul N Whatmough, Sae Kyu Lee, Hyunkwang Lee, Saketh Rama, David Brooks, and Gu-Yeon Wei. 14.3 a 28nm soc with a 1.2 ghz 568nj/prediction sparse deep-neural-network engine with¿ 0.1 timing error rate tolerance for iot applications. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 242–243. IEEE, 2017.

[113] Jeremy HM Wong, Yashesh Gaur, Rui Zhao, Liang Lu, Eric Sun, Jinyu Li, and Yifan Gong. Combination of end-to-end and hybrid models for speech recognition. In *Interspeech*, pages 1783–1787, 2020.

[114] Hainan Xu, Tongfei Chen, Dongji Gao, Yiming Wang, Ke Li, Nagendra Goel, Yishay Carmiel, Daniel Povey, and Sanjeev Khudanpur. A pruned rnnlm lattice-rescoring algorithm for automatic speech recognition. In *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5929–5933. IEEE, 2018.

[115] Qiantong Xu, Alexei Baevski, Tatiana Likhomanenko, Paden Tomasello, Alexis Conneau, Ronan Collobert, Gabriel Synnaeve, and Michael Auli. Self-training and pre-training are complementary for speech recognition. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3030–3034. IEEE, 2021.

[116] Xuerui Yang, Jiwei Li, and Xi Zhou. A novel pyramidal-fsmn architecture with lattice-free mmi for speech recognition. *arXiv preprint arXiv:1810.11352*, 2018.

[117] Reza Yazdani, Jose-Maria Arnau, and Antonio González. Unfold: A memory-efficient speech recognizer using on-the-fly wfst composition. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–81, 2017.

[118] Reza Yazdani, Jose-Maria Arnau, and Antonio González. Laws: Locality-aware scheme for automatic speech recognition. *IEEE Transactions on Computers*, 69(8):1197–1208, 2020.

[119] Reza Yazdani, Marc Riera, Jose-Maria Arnau, and Antonio González. The dark side of dnn pruning. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 790–801. IEEE, 2018.

[120] Reza Yazdani, Albert Segura, Jose-Maria Arnau, and Antonio Gonzalez. An ultra low-power hardware accelerator for automatic speech recognition. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[121] Dong Yu and Li Deng. *Automatic Speech Recognition.* Springer, 2016.

[122] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. *ACM SIGARCH Computer Architecture News*, 45(2):548–560, 2017.

[123] Neil Zeghidour, Qiantong Xu, Vitaliy Liptchinsky, Nicolas Usunier, Gabriel Synnaeve, and Ronan Collobert. Fully Convolutional Speech Recognition. *arXiv e-prints*, page arXiv:1812.06864, Dec 2018.

[124] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[125] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, Jian Tang, Wujie Wen, Makan Fardad, and Yanzhi Wang. A systematic dnn weight pruning framework using alternating direction method of multipliers. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 184–199, 2018.

[126] Yu Zhang, James Qin, Daniel S Park, Wei Han, Chung-Cheng Chiu, Ruoming Pang, Quoc V Le, and Yonghui Wu. Pushing the limits of semi-supervised learning for automatic speech recognition. *arXiv preprint arXiv:2010.10504*, 2020.

Glossary

AM Acoustic Model, one of the component of speech recognition systems. The acoustic model processes the audio frames to compute a probability distribution over acoustic units, such as phonemes or characters..

ASR Automatic Speech Recognition, the conversion of a speech signal to a symbolic representation by computational means.

CONV A type of neural network layer. It performs a convolution over the input.

CTC Connectionist Temporal Classification, a type of end-to-end speech recognition system.

CU Cumputation Unit, common name for the hardware components that perform arithmetic operations in an accelerator.

DNN Deep Neural Network, computational system consisting of a sequence of interconnected layers that perform a non-linear transformation to their inputs.

end-to-end Speech recognition system whose acoustic model is just a neural network.

FC Fully-connected, a type of neural network layer. It consists of a set of neurons that are all connected to all the outputs from the previous layer.

HCLG Common decoding graph used by hybrid ASR systems. Is the result of combining 4 graphs: HMM, Context-dependency, Lexicon and Grammar.

HMM Hidden Markov Model, mathematical tool to model markov processes. In the context of speech recognition, HMMs can be used to model acoustic units, such as phonemes or triphones.

HMM-GMM Speech recognition system that contains Hidden Markov models and gaussian Mixture Models to model acoustic units.

hybrid DMM-DNN Speech recognition system whose acoustic model is a combination of Hidden Markov Models and a neural network.

hypothesis a sequence of words or acoustic units generated by a speech recognition system as the result of decoding an input signal.

label another name for ”acoustic unit”. It is common in the literature related to hybrid and CTC speech recognition.

lattice Graph that contains a set of possible transcriptions.

LM Language Model, one of the components of speech recognition systems. The Language model assigns a probability to each sequence of words.

MAC Multiply-and-accumulate, mathematical operation that multiplies two of its operands together and accumulates the result with a third operand.

MFCC Mel-Frequency Cepstrum Coeficients, a feature vector to represent audio frames. It is computed essentially by transforming the signal into the frequency domain, restricted by the range of audible frequencies and then weighting each frequencies according to the Mel-scale, which closely mimics the frequency sensitivity of the human ear.

PE Processing Elements, common name for the hardware components that perform arithmetic operations in an accelerator.

ReLU Rectified Linear Unit. Activation function used in neural network. It clips negative numbers to zero while leaving positive inputs unchanged.

seq2seq A type of end-to-end speech recognition system.

Streaming ASR performing automatic speech recognition in batches of inputs, instead to capturing the entire signal and decoding it at once.

TDNN In the context of this thesis, a hybrid speech recognition system whose acoustic model is a Time-Delay Neural Network.

TDS In the context of this thesis, an end-to-end speech recognition system whose acoustic model is a neural network built composed of Time-Depth Separable convolution blocks.

Viterbi Beam Search Algorithm to find the most likely path through a Hidden Markov Model given a sequence of observations.

WER Word Error Rate, common metric to measure accuracy of speech recognition systems. It is the ratio of word errors divided by the number of words in the ground truth.

WFST Weighted Finite-State Transducer, a type of graph that represents an algorithms to transform a sequence from an input vocabulary to an output vocabulary. For example, a lexicon WFST will transform a sequence of phonemes into a sequence of words.

word embeddings a vector representation of words. These vectors are considered to be part of a high dimensional space where the distance among them is a measure of the how related the words are. Each word is assigned a word embedding by an optimization algorithm.

word-pieces acoustic unit for speech recognition systems. Each word-piece is a arbitrary sequence of letters obtained by breaking down the words from a vocabulary following an optimization algorithm.