POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Accelerating Transformer Inference on Heterogeneous Multi-Accelerator SoCs using ESP

Supervisors

Prof. Daniele Jahier PAGLIARI

Dr. Alessio BURRELLO

Prof. Luca CARLONI

Dr. Mohamed Amine HAMDI

Candidate

Jessica MAROSSERO

Academic Year 2023-2024

Summary

Transformers have become essential in deep learning, excelling in tasks like natural language processing and computer vision. However, they are computationally expensive, especially in their so-called attention layers, which require large-scale matrix multiplications with quadratic complexity. Therefore, coupling general purpose processors with specialized hardware accelerators is critical to efficiently deploy Transformers in embedded systems with limited resources. The Embedded Scalable Platform (ESP) is a pioneering open-source research platform that enables the design of such heterogeneous SoCs, by integrating multiple types of tiles in a 2D mesh architecture. This modular design allows for an efficient integration of third-party accelerators, enabling rapid prototyping and exploration of novel architectures.

This thesis focuses on the integration of the state-of-the-art Integer Transformer Accelerator (ITA) within ESP. ITA was developed to accelerate the execution of Transformer models by employing 8-bit quantization and custom hardware optimizations to improve the efficiency and reduce the memory footprint of attention, including an efficient implementation of the softmax function, a key component of this type of layer. Its integration in ESP required incorporating private local memories (PLMs) within ITA to store data locally during computation, minimizing external memory access. A controller was designed to manage DMA transactions, ensuring efficient data movement between the system memory and the PLMs. Additionally, a hardware socket was generated to interface ITA with the ESP platform. The latter facilitates communication between the accelerator and system components, allowing ITA to be integrated seamlessly into the SoC architecture.

The final SoC architecture then consists of a memory tile, an Ariane RISC-V CPU tile, an ITA tile and an I/O tile.

On the software side, a bare-metal application was written to validate the functionality of ITA within the ESP system. This application demonstrated the capability of ITA to perform attention computations taken from a real-world transformer model, working in coordination with the Ariane processor.

The results showed significant improvement when using ITA to accelerate attention layers, with respect to a purely software solution running entirely on Ariane.

Lastly, the flexibility of ESP was leveraged to explore performance scalability, by increasing the number of ITA accelerator tiles, which each processing a different attention head in parallel.

In summary, this thesis demonstrates the successful design and integration of a specialized hardware accelerator for transformer models, exploiting the flexibility and modularity of ESP. The final SoC represents a promising solution for the deployment of resource-intensive machine learning models in embedded-systems.

Acknowledgements

I would like to express my sincere gratitude to my parents, who have supported me throughout my academic journey. Their sacrifices and constant support have allowed me to take advantage of many precious opportunities and experiences during these years, and for this I will always be deeply grateful.

I would also like to extend my gratitude to my friends and all the people who have been by my side during my entire journey. Their support has helped me believe in myself, overcome many difficulties, and reminded me to be a bit positive sometimes.

Lastly, I would like to thank my supervisors, Prof. Daniele Jahier Pagliari, Dr. Alessio Burrello, Dr. Mohamed Amine Hamdi, and Prof. Luca Carloni, for giving me the opportunity to work on this interesting project. Their guidance, constant availability and precious assistance have been crucial to my research progress and I am very grateful for their support.

Table of Contents

List of Tables VIII

List of Figures IX

Acronyms XII

1 Introduction 1

2 Background 4

2.1 Deep Neural Networks 4
2.2 Multi-Layer Perceptron 4
2.3 RNNs and LSTMs 6
2.4 Transformers 6
2.4.1 Encoder 8
2.4.2 Decoder 8
2.4.3 Positional Encoding 9
2.4.4 Attention 9
2.4.5 Feed-Forward Networks 12
2.5 Deep Learning Accelerators 13
2.5.1 Transformer Accelerators 15
2.6 HW Platform 16
2.6.1 ITA: Integer Transformer Accelerator 16
2.6.2 ESP 20
2.6.3 ESP accelerator specification 23

3 Accelerating Attention on ESP 28

3.1 Integration of one ITA instance into ESP 29
3.1.1 PLMs management 29
3.1.2 DMA Controller Design and Implementation 34
3.1.3 Accelerator Integration 37
3.1.4 SoC Generation and Simulation 43

3.1.5 FPGA Deployment 45
3.2 Integrating Multiple ITA Instances: Multi-Head Attention Support 46
3.2.1 Multi-heads attention: sequential vs parallel 46
3.2.2 Finalizing Layer Acceleration in Transformer Encoder 54
3.3 Validation of Accelerator Performance 59

4 Experimental Results 61
4.1 Baseline Software Performance 61
4.1.1 The Need for Hardware Acceleration 62
4.2 ITA Acceleration: Performance Evaluation and Analysis 63
4.3 Optimizing the Attention Mechanism with Multiple Accelerators . . 69
4.3.1 Parallelization Strategies: Sequential vs. Parallel Execution . 69
4.3.2 Two Heads: Sequential vs. Parallel Execution 69
4.3.3 Four Heads: Sequential vs. Parallel Execution 70
4.3.4 Reduced Benefits as Parallelization Increases 72
4.3.5 Pareto Optimal Trade-offs Between Latency and Area 73

5 Conclusions and future works 75

Bibliography 78

List of Tables

3.1 Encoding of DMA size 36
3.2 Memory Map and base pointers 40

4.1 Execution Time Breakdown for a complete Transformer Layer . . . 62
4.2 Execution Time Breakdown for a complete Transformer Layer . . . 65
4.3 Two Heads - Sequential Execution (1 ITA accelerator) 70
4.4 Two Heads - Parallel Execution (2 ITA accelerators) 70
4.5 Four Heads - Sequential Execution (1 ITA accelerator) 71
4.6 Four Heads - Parallel Execution (2 ITA accelerators) 71
4.7 Four Heads - Fully Parallel Execution (4 ITA accelerators) 72

List of Figures

2.1 A 2-layer Neural Network: one input layer with three inputs, one hidden layer of 4 neurons and one output layer with 2 neurons [8] . 5
2.2 The Transformer model architecture [1] 7
2.3 (Left) Scaled Dot-Product Attention (Right) Multi-Head Attention [1] 10
2.4 Transformer encoder and multi-head attention [5] 11
2.5 Spatial and temporal architectures, taken by [17] 14
2.6 Architecture of ITA with 8-bit inputs and weights [5] 16
2.7 Tiling and computation phases [5] 17
2.8 HWPE interface [29] 19
2.9 Data requirement 19
2.10 Design flows supported by ESP [7] 20
2.11 Block diagram of the ESP accelerator model [35] 24
2.12 Example of a DMA read transaction in ESP [35] 26
2.13 Example of a DMA write transaction in ESP [35] 27

3.1 HWPE with ITA 29
3.2 Block diagram of ITA accelerator tile in a 2×2 ESP SoC 39
3.3 ESP GUI with one ITA accelerator 44
3.4 Debug Link of the ESP GUI 44
3.5 ESP GUI with two ITA accelerators 50

4.1 Time breakdown for a complete Transformer layer in all its operations 61
4.2 Time breakdown for a complete Transformer layer in Attention layer and addition and normalization 62
4.3 ITA acceleration 64
4.4 Execution breakdown for the total layer with ITA and Ariane processor 64
4.5 Layer acceleration: Ariane vs ITA + Ariane 65
4.6 Area breakdown showing the contributions of Ariane processor and ITA accelerator in FPGA deployment. 66
4.7 FPGA resource allocation 67
4.8 Resource breakdown of the ITA accelerator on FPGA 68

4.9 Pareto Curve: Latency vs. Area for Multi-Head Attention with 4 Heads 73

Acronyms

ESP
: Embedded Scalable Platform

ITA
: Integer Transformer AcceleratoR

PLM
: Private Local Memory

DMA
: Direct Memory Access

SoC
: System-On-Chip

CPU
: Central Processing Unit

RNN
: Recurrent Neural Network

LSTM
: Long Short-Term Memory networks

NLP
: Natural Language Processing

BERT
: Bidirectional Encoder Representations from Transformers

GPT

Generative Pre-trained Transformer

ViT

Vision Transformer

AI

Artificial Intelligence

CNN

Convolutional Neural Network

HLS

High-Level Synthesis

DNN

Deep Neural Network

NoC

Network-on-Chip

ML

Machine Learning

DL

Deep Learning

ANN

Artificial Neural Network

MLP

Multi-Layer Perceptron

ReLU

Rectified Linear Unit

MHSA

Multi-Head Attention

FFN

Feed-Forward Network

LN

Layer Normalization

GELU

Gaussian Error Linear Unit

MAC

Multiply-and-ACcumulate

GPU

Graphic Processing Unit

FPGA

Field Programmable Gate Array

ASIC

Application Specific Integrated Circuit

PE

Processing Element

TPU

Tensor Processing Unit

HWPE

HardWare Processing Engines

PULP

Parallel Ultra Low Power

TCDM

Tightly Coupled Data Memory

FIFO

First-In-First-Out

FSM

Finite State Machine

GEMM

General Matrix Multiplication

Chapter 1

Introduction

In recent years, Transformers have emerged as an innovative architecture in Deep Learning. Initially proposed by Vaswani et al. in the year 2017[1], Transformers were intended to tackle the challenges that come with the sequential approach of Recurrent Neural Networks (RNNs) and Long Short-Term Memory networks (LSTMs). Considered as an improvement over RNNs and LSTMs, Transformers rely on an attention mechanism to assess the relevance of different parts of an input sequence and perform Natural Language Processing (NLP) applications like translation, text generation and language comprehension effectively. In addition, the strength of Transformers has been seen in areas of computer vision, speech recognition and biological sequence analysis among other applications other than NLP.

BERT[2], GPT[3], and Vision Transformer (ViT)[4] are among the most popular Transformer model architectures. These specific models have pioneered the advancement of machine learning, reaching record levels of precision in several types of applications, beginning with NLP and ending with image recognition. BERT (Bidirectional Encoder Representations from Transformers) introduced a bidirectional training method to capture context from both directions of a sentence. This resulted in cutting-edge performance in areas such as question-answering and language inference. Conversational Artificial Intelligence (AI) has improved dramatically through GPT's autoregressive architecture (Generative Pre-trained Transformer), demonstrating impressive skills in text generation and improving natural language understanding. The advancement of GPT models represented major achievements in increasing the size of Transformer models to billions of parameters, resulting in their ability to produce coherent, human-like text.

Even models, like ViT[4], have demonstrated outstanding performance on large datasets, compared to the achievement of image classification with Convolutional Neural Networks (CNNs). However, this success entails considerable expenses, as Transformers impose heavy demands on both memory size and computational

power if attention mechanisms are to be implemented. Their real-time operation in systems, especially with limited resources such as embedded systems, becomes increasingly difficult, as there are billions of parameters to deal with. The extreme computational requirements led to increased interest in devices that could serve as efficient hardware accelerators for such models.

To address these challenges, the Integer Transformer Accelerator (ITA)[5] was developed as a hardware accelerator designed to optimize the execution of the Transformer attention layer in low-power environments. ITA leverages 8-bit quantization and an optimized approach to the softmax function. By using 8-bit integers instead of 32-bit floating-point numbers, ITA reduces the amount of data processed, saving both compute power and memory bandwidth.

One of the key innovations in ITA is the integer-based softmax implementation, which eliminates the need for expensive floating-point units. Compared to its base implementation, which computes the exponentials of the input values, sums these factors, and then divides them by the sum. This involves multiple floating-point multiplications and divisions which are computationally expensive. While the integer-based softmax, not only reduces hardware complexity but also minimizes data movement, as the non-linear function is computed in a streaming fashion.

As Transformers continue to dominate the Machine Learning field, the integration of specialized hardware accelerators like ITA is crucial for efficiently deploying these models in real-world applications, and, platforms such as the Embedded Scalable Platform (ESP)[6][7] are essential to meet growing computing demands.

ESP is a pioneering open-source platform for heterogeneous System-on-Chip (SoC) design. It supports rapid prototyping and deployment of hardware accelerators like ITA, within a modular, tile-based architecture. This platform is suited for applications spanning from embedded systems to high-performance computing and supports accelerators designed in languages such as C/C++, SystemC, Verilog, or High-Level Synthesis (HLS) tools.

ESP's modular design integrates various tiles, including compute, memory, and accelerator tiles, using a Network-on-Chip (NoC) for high-bandwidth communication. This flexibility allows ITA to be seamlessly integrated with other processing elements, optimizing AI inference. The platform's scalability and ability to handle diverse workloads make it an ideal solution for exploring configurations in resource-constrained systems.

The focus of this thesis relies in integrating the ITA Transformer accelerator into ESP to enhance the performance of Transformer attention mechanisms. This research was conducted in collaboration with Columbia University, particularly with the SLD group, which developed the ESP platform. The project involved adapting ITA's interface for compatibility with ESP's modular system, configuring Private Local Memories (PLMs), designing a custom Direct Memory Access (DMA) controller for efficient data transfers, and developing a bare-metal application to

validate ITA's performance.

The thesis is organized into five chapters. The first chapter provided an introduction to Transformers, ITA, and ESP. The second chapter covers the background necessary to understand the work done in this thesis, including an introduction to Deep Neural Networks (DNNs), theoretical concepts of Transformers, their architecture, key mathematical operations, and an overview of Deep Learning accelerators, with a focus on Transformer accelerators. It also describes the hardware platforms used. The third chapter details the work undertaken to integrate the ITA accelerator into the ESP platform, outlining the important steps and actions required to replicate the work. The fourth chapter presents the results obtained, emphasizing the need for dedicated hardware to achieve faster execution speeds. The final chapter summarizes the conclusions on the project and outlines potential future work that could build upon this research.

Chapter 2

Background

2.1 Deep Neural Networks

Machine Learning (ML) has significantly transformed various fields by enabling systems to learn from data rather than just depending on explicit programming. Traditional ML methods often require manual feature engineering, with experts creating features considered to be most suitable for the specific task.

Deep Learning (DL) has further advanced ML by integrating feature extraction and representation learning into a single, cohesive process. Deep Neural Networks are the key models that distinguish DL. DNNs differ from traditional ML methods by being able to extract meaningful features from raw data without the need for human-created features during training. This ability comes from their structured design, with each level extracting and improving characteristics from the previous ones.

DNNs are structured as layered graphs, with each layer performing specific operations to transform the input data. Through training, these networks adjust their parameters to optimize performance on a given task.

2.2 Multi-Layer Perceptron

A Multi-Layer Perceptron (MLP) is a type of Artificial Neural Network (ANN) consisting of multiple layers of neurons. The MLP architecture typically consists of three types of layers, as you can see in Fig. 2.1: the input layer, one or more hidden layers, and the output layer. The input layer receives the raw data and passes it to the network. The hidden layers, which sit between the input and output layers, are where the actual learning happens, as each hidden layer applies transformations to the data. The output layer generates the final predictions or classifications based on the processed data. The depth of the network (i.e. the number of hidden layers)

and the number of neurons in each layer determine the MLP's capacity to learn complex functions.

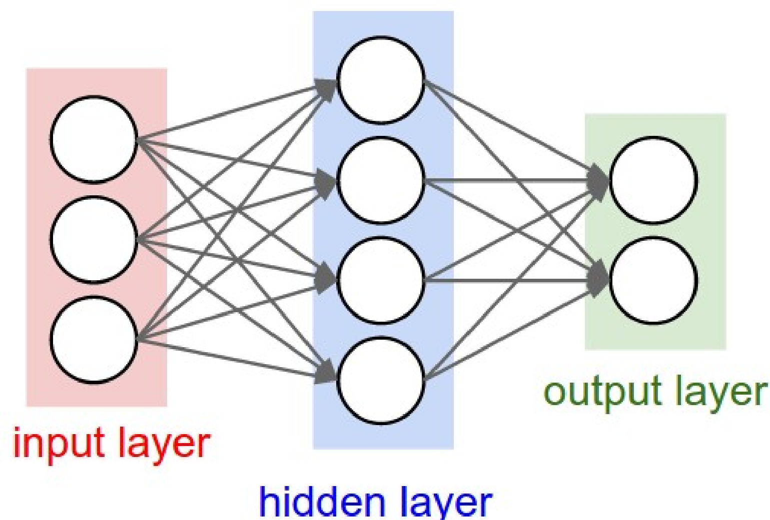

Figure 2.1: A 2-layer Neural Network: one input layer with three inputs, one hidden layer of 4 neurons and one output layer with 2 neurons [8]

Each neuron performs a mathematical operation inspired by the functioning of biological neurons in the human brain. The artificial one, which is a linear transformation, consists of n inputs x_i and one output y. The output is calculated as follows:

$$y = f\left(\sum_{i=1}^{n} x_i \cdot w_i + b\right)$$

where f is the activation function, w_i are the weights, and b is the bias term. The activation function introduces non-linearity to the neuron's output, which is crucial for the network's ability to learn complex patterns. One of the most commonly used activation functions is ReLU, defined as:

$$\mathrm{ReLU}(x) = \max(0, x)$$

Rectified Linear Unit (ReLU) activates neurons only if the input is positive, setting all negative values to zero while leaving positive values unchanged.

For more details on DNN structure, training, loss, and optimizers, refer to the book of Ian Goodfellow and Yoshua Bengio[9].

2.3 RNNs and LSTMs

While Multi-Layer Perceptrons have demonstrated remarkable capabilities in handling various tasks, they are not well-suited for sequential data due to their fixed-size input and inability to capture temporal dependencies. This limitation led to the development of RNNs, which are specifically designed to process sequences by maintaining hidden states that evolve over time as new inputs are received. RNNs can be thought of as neural networks with loops, allowing information to be passed from one step to the next, thus capturing dependencies across time. However, standard RNNs often struggle with long-range dependencies, primarily due to the vanishing gradient problem [10] [11], which makes it challenging for the network to learn relationships between distant elements in a sequence.

To address these limitations, LSTMs were introduced, incorporating memory cells and gating mechanisms. The architecture of LSTMs enables them to selectively remember or forget information, thus mitigating the issues associated with standard RNNs and making them more effective for tasks such as language modeling, speech recognition, and time series forecasting. For more information, see [12], [13], and [10].

Although there have been improvements, RNNs and LSTMs still operate sequentially, causing inefficiencies in both training and inference, particularly with long sequences. As a result, the emergence of Transformers represented a major advancement in the field of Deep Learning.

2.4 Transformers

The Transformer is a deep neural network model first introduced by Vaswani et al in 2017 [1], and it is largely based on the attention mechanism of modeling.

This architecture was aimed to handle data that can be represented as sequences (or collections of tokens where the order is important). Such approaches were originally predicted to resolve certain types of tasks related to natural language (translating and text generation). The transformer architecture has also proved useful in other challenges such as computer vision and multi-modal learning. The model's effectiveness, especially in tasks involving recognition of patterns over long sequences has made it an everyday model employed in many machine learning problems.

As shown in Fig. 2.2, the Transformer model is composed of two main parts: the encoder, located on the left side of the Figure, and the decoder, situated on the right side. This structure reflects its original application in machine translation. The encoder processes a source sentence and finds a latent representation, while the decoder uses this representation to reconstruct an equivalent target sentence,

thus enabling translation between languages.

More recent works leverage only the encoder element for vision tasks. The encoder serves as a feature extractor, while an MLP head is added to perform tasks like classification, and processing the features extracted from the input.

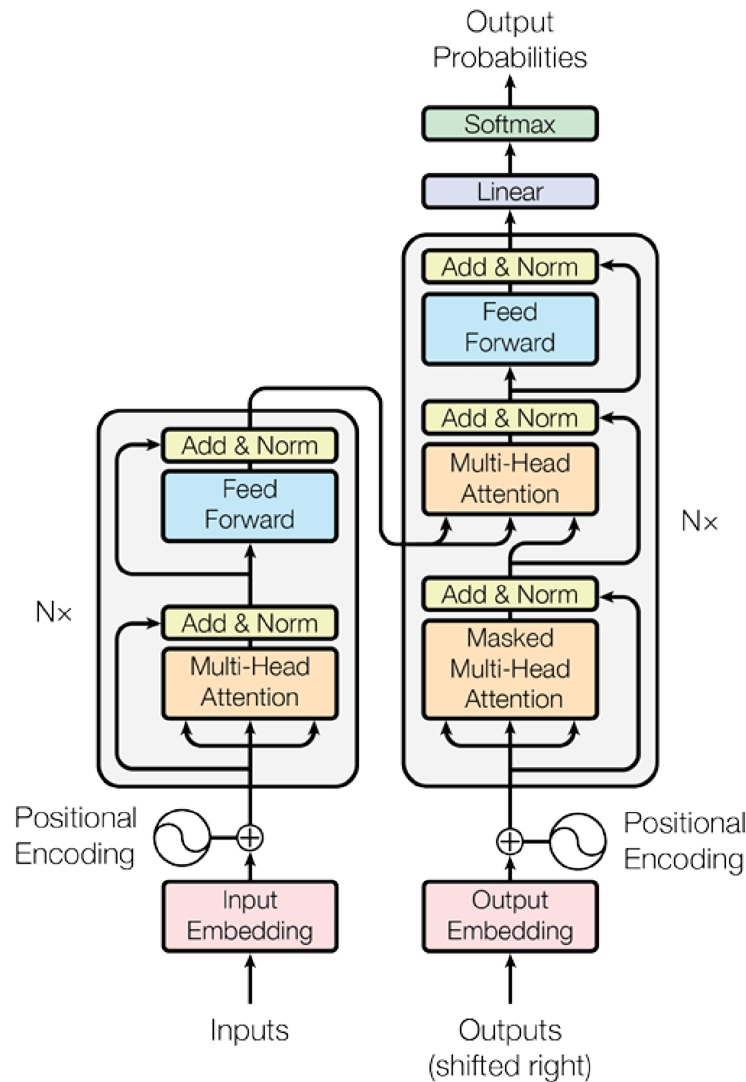

Figure 2.2: The Transformer model architecture [1]

Transformers have gained significant attention due to their benefits over previous architectures, especially Recurrent Neural Networks:

- **Parallelization**: Unlike RNNs, which involve a sequential step of one-token

processing at each step, in Transformer encoder structures, all parts of a sequence can be simultaneously treated due to the use of the attention layer. This can happen because the outcome of the attention is not influenced by any state variable. RNNs suffer from a bottleneck as they require the output from the previous step before proceeding to the next.

- **Long-range dependencies**: RNNs frequently have difficulty capturing long-term dependencies, as data can deteriorate or disappear with time. Transformers utilize self-attention mechanisms to capture information.

However, Transformers also come with some limitations:

- **Data requirements**: Transformers often require large datasets to train effectively. This makes them less suitable for tasks with limited labeled data than simpler models that perform well with smaller datasets.
- **Positional encoding**: Since Transformers process input sequences in parallel, they lack the inherent ability to capture positional information that RNNs have. To address this, they rely on additional positional encodings, which may not always be as effective in capturing order information as RNNs.

2.4.1 Encoder

The Transformer encoder consists of N identical layers, each composed of two main components: a Multi-Head Attention (MHA) mechanism and a feed-forward network (FFN). The MHA allows the model to focus on different parts of the input sequence simultaneously, capturing both local and global dependencies and the MLP head embeds each encoding.

Each layer in the encoder uses residual connections and Layer Normalization (LN) to stabilize the training process and maintain learning efficiency[14]. Each layer can be described as:

$$\tilde{X}_{l+1} = \mathrm{LN}(X_l + \mathrm{MHA}(X_l))$$

$$X_{l+1} = \mathrm{LN}(\tilde{X}_{l+1} + \mathrm{MLP}(\tilde{X}_{l+1}))$$

2.4.2 Decoder

In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Just like the encoder, it uses residual connections around every sub-layer, along with layer normalization. In the decoder stack, changes are made to the self-attention sub-layer so that positions are restricted from attending to future

positions. This masking, along with the condition that the output embeddings are shifted by a single position, guarantees that the predictions for position i are only influenced by the known results at positions preceding i [4].

The decoder is briefly described here for completeness, even if this thesis will focus on encoder-only Transformers.

2.4.3 Positional Encoding

The Transformer architecture processes tokens simultaneously, lacking the inherent sequential layout characteristic of RNNs. To compensate for this, the model incorporates input embeddings that transform each token into a continuous vector space, allowing it to effectively represent the semantic information of the tokens. However, since the simultaneous processing of tokens can obscure their positional relationships, the Transformer integrates positional encodings into these input embeddings.

Positional encoding provides the model with information regarding the relative and absolute positions of each token within the sequence. A common approach for generating these encodings utilizes sinusoidal waveforms, which enable the model to generalize to longer sequences that may not have been encountered during training. This design not only allows the Transformer to recognize the significance of position but also enhances its ability to adapt to varying sequence lengths while maintaining contextual understanding.

2.4.4 Attention

The central operation in both the encoder and decoder layers of the Transformer model is the scaled dot-product attention. This operation can be repeated in parallel in several instances, called heads, resulting in what is known as multi-head attention, as you can see in Fig. 2.3. The figure depicts, on the left, the scaled dot-product attention, and on the right, the multi-head attention. They will be explained in detail in the following sections.

The attention operation can be defined as a function that, when given a query vector and a set of key-value pairs, produces a weighted sum of the values as output. The weights are calculated by determining a compatibility score for each key with the query.

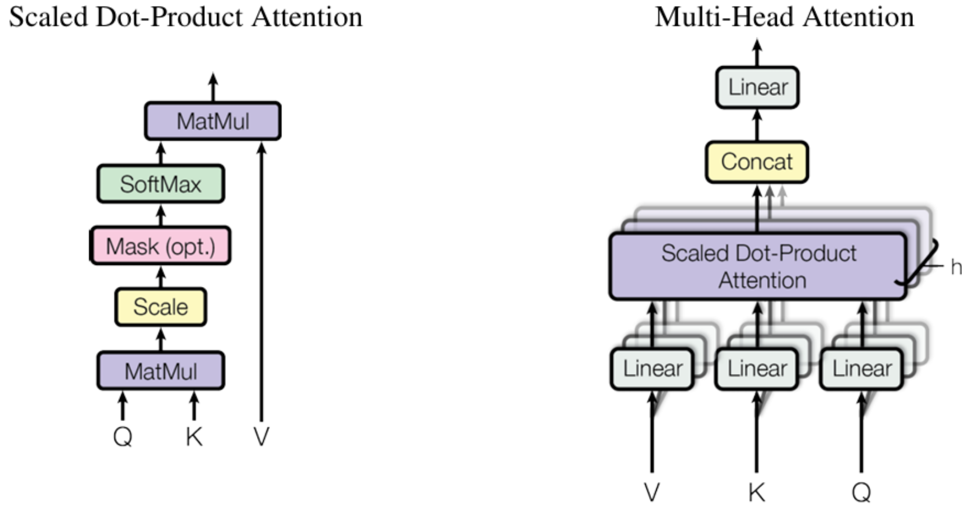

Figure 2.3: (Left) Scaled Dot-Product Attention (Right) Multi-Head Attention [1]

Scaled Dot-Product Attention

In standard Transformer models, the attention function used is the Scaled Dot-Product Attention, which operates on three inputs: the query (Q), key (K), and value (V) matrices. The function is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where Q, K, and V are obtained by applying linear transformations to the original inputs of size $S \times E$, resulting in matrices of size $S \times P$, with S being the sequence length, E the embedding size, and P the projection space, as illustrated in Fig. 2.4.

The attention computation proceeds by performing the matrix multiplication between Q and K^T, followed by dividing the result by the square root of the dimensionality of the keys, d_k, to scale the similarity scores. The softmax function is then applied to generate an $S \times S$ attention matrix A, which represents the probabilities. This matrix is subsequently multiplied by V to produce the output O of size $S \times P$.

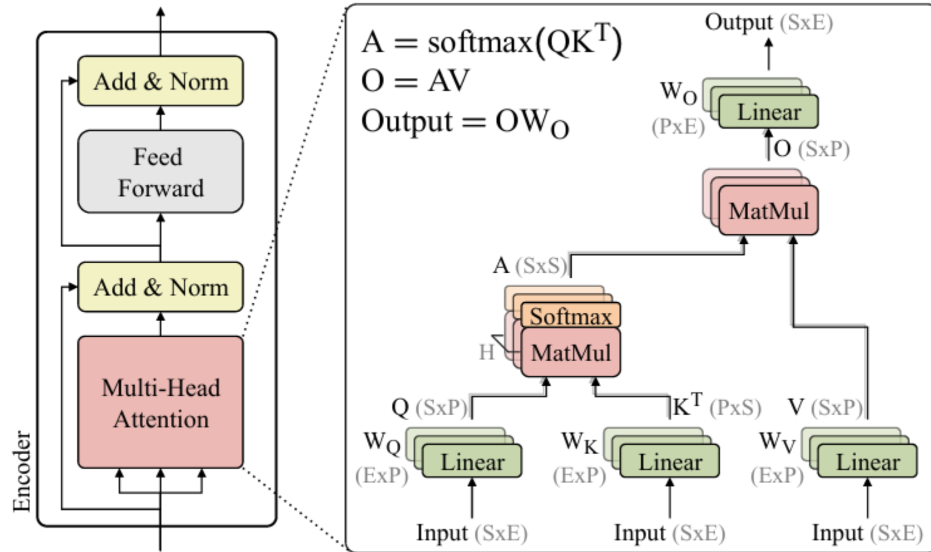

Figure 2.4: Transformer encoder and multi-head attention [5]

Softmax Function

The Softmax function is a crucial component in the Scaled Dot-Product Attention mechanism used in Transformers. The Softmax function converts a vector of values into a probability distribution by exponentiating the values and then normalizing them. The Softmax function is defined as:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

where x_i represents the i-th element of the input vector, and the denominator is the sum of the exponentials of all elements in the vector. This normalization ensures that the output values lie between 0 and 1 and sum up to 1, making them interpretable as probabilities.

In the context of Scaled Dot-Product Attention, the Softmax function is applied to the scaled dot-product scores. By doing this, the Softmax function converts these scores into a probability distribution over the keys, which determines the weighting of the values. This ensures that the attention mechanism focuses on the most relevant parts of the input sequence while providing a clear, probabilistic interpretation of the attention weights.

Softmax in the attention mechanism is crucial for handling varying magnitudes of the dot product scores and ensuring numerical stability, as it prevents the model

from assigning excessively high weights to any single key or value.

Multi-Head Attention

Multi-Head Attention enables the model to simultaneously focus on data from various subspaces of representation at varying positions. MHA is composed of multiple attention layers, or heads, operating simultaneously. Every head applies a distinct linear projection on the queries, keys, and values that have been learned before using the attention function.

$$\text{MultiHead}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

Each attention head is computed as:

$$\text{head}_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^V) = \text{Attention}(Q, K, V)$$

In this case, X represents the input matrix, and W_i^Q, W_i^K, and W_i^V represent trained projection matrices for the i-th attention head. The final output is produced by concatenating and projecting the outputs of the various attention heads with the matrix W^O. This enables the model to focus on multiple parts of the sequence at the same time, improving its capability to understand intricate relationships between tokens.

Once the attention outputs are calculated, they go through a residual connection and layer normalization to ensure that the network can still learn well as more layers are added.

2.4.5 Feed-Forward Networks

The Feed-Forward Network module in the Transformer consists of two linear transformations separated by a non-linearity. Following the processing of the input sequence by the multi-head attention mechanism, the output undergoes transformation through it. Each of the layers in the encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically.

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

While the original Transformer employed the ReLU activation function, more recent models, particularly those in vision applications, frequently opt for the Gaussian Error Linear Unit (GELU) activation function over ReLU. The GELU[15] function, as introduced by Hendrycks and Gimpel, blends the advantages of a linear unit and a non-linear activation to offer a more seamless activation.

$$\text{GELU}(x) = x \cdot \Phi(x) = \frac{x}{2}\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

$\Phi(x)$ is the cumulative distribution function of the standard normal distribution. The error function, $\mathrm{erf}(x)$, is a mathematical function defined as:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

It is used to provide a smooth approximation to the ReLU function and it introduces a probabilistic smoothing effect, which can improve gradient flow. Unlike ReLU, which simply thresholds negative values to zero, GELU allows a small number of negative values to pass through, leading to improved model performance in various tasks.

2.5 Deep Learning Accelerators

Convolutional Neural Networks and Transformers in deep learning require large computational resources because they involve a high number of matrix computations, typically including Multiply-and-ACcumulate (MAC) [16] operations. The fundamental computational task in convolutional and fully connected layers is the MAC operation, which consist of calculating dot products between input features and model parameters repeatedly. Modern deep learning models require millions, if not billions, of these operations.

Specialized hardware accelerators have been proposed to handle the computational needs of deep learning by focusing on specific tasks in neural networks. There are two primary architectural paradigms used for accelerating deep learning computations: temporal architectures and spatial architectures[17]:

- **Temporal architectures**, used by Central Processing Units (CPUs) and Graphic Processing Units (GPUs), rely on centralized control for multiple Arithmetic-Logic Units (ALUs) that operate independently without exchanging data between each other. They retrieve data from memory, execute required operations, and subsequently store the outcomes back in memory. Although temporal architectures are versatile and flexible, they may not be as efficient when it comes to reusing data and facilitating communication between ALUs, potentially causing congestion in memory access and data transfer.
- **Spatial architectures**, on the other hand, are frequently utilized in Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs). In this design, the ALUs are placed in an array and can communicate with each other, enabling improved data recycling and increased throughput. This specific architectural design is highly appropriate for deep learning accelerators as it reduces the need for memory access and allows for detailed parallel processing at the hardware level.

As you can see in Fig. 2.5, spatial and temporal architectures have a similar computational structure, with a set of Processing Elements (PEs). However, processing units can have internal control in spatial architecture, whereas control in temporal architecture is centralized. Each PE can have a register file (RF) to store data in spatial architecture, however, PEs do not have a dedicated memory capacity in a temporal architecture.

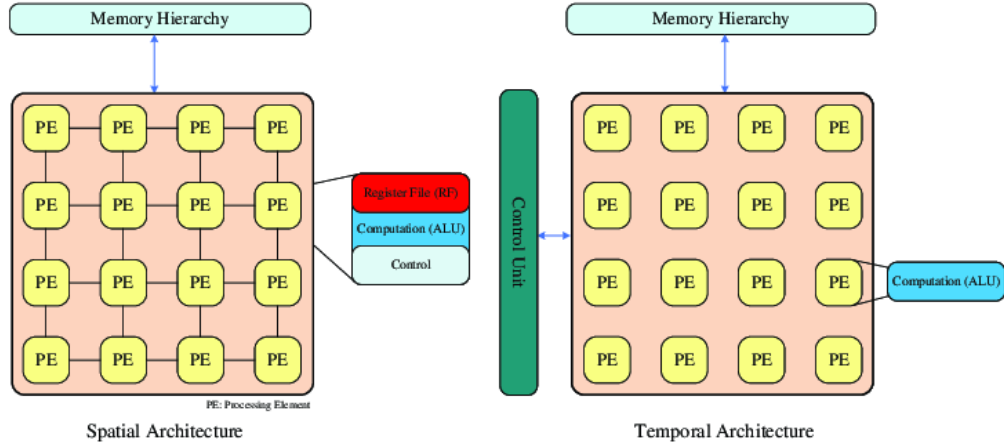

Figure 2.5: Spatial and temporal architectures, taken by [17]

In the early stages of hardware acceleration, the development of hardware accelerators focused on temporal architectures, primarily using CPUs and GPUs. CPUs, with their general-purpose architecture, were first employed to execute deep learning tasks. However, due to their limited parallelism and efficiency for large-scale matrix operations, GPUs soon became the preferred choice.

Due to their high throughput and memory bandwidth, GPUs are frequently used hardware accelerators to enhance inference and training processes in DNNs [18]. On the other hand, GPU-based hardware accelerators consume a lot of power. ASIC and FPGA-based hardware accelerators have limited computational and memory resources in comparison to GPU accelerators. They can, nevertheless, achieve a moderate level of performance while using less energy [19]. ASIC-based DNN accelerators offer better performance than GPU and FPGA options. However,ASIC-based accelerators have some limitations, including high cost of development, long time to market, inflexibility, etc. FPGA-based accelerators can be used as an alternative to ASIC-based accelerators, and they can provide superior performance at an affordable cost with reconfigurability and low power dissipation [20].

Also the introduction of Tensor Processing Units (TPUs) represented a significant advancement in hardware acceleration. TPUs are custom-designed hardware accelerators specifically developed by Google to handle the high computational demands of deep learning models. Introduced by Jouppi et al.[21], TPUs are

optimized for matrix operations and tensor computations, offering substantial performance improvements over traditional GPUs for both training and inference tasks.

More detailed information on FPGA, ASIC, and GPU-based accelerators for deep learning can be found in the works by Dhilleswararao et al. [17], Jouppi et al. [21], Silvano et al. [22], and also in [23], [24], [25].

2.5.1 Transformer Accelerators

Accelerating inference of Transformer networks is an active area of research, with most accelerators focusing on the attention layer. Since Transformers divide the attention mechanism into multiple heads, hardware accelerators can exploit this inherent parallelism by processing each head simultaneously. This requires careful management of data dependencies and synchronization between the heads to ensure efficient computation.

The design of these hardware accelerators shares many similarities with CNN accelerators but requires additional optimizations to handle the unique demands of the attention mechanism.

In recent years, several innovative hardware accelerators specifically designed for Transformers have emerged. Some architectures exploit the sparsity of the attention matrix, such as OPTIMUS, SpAtten, and ELSA.

OPTIMUS, as detailed in the paper by Park et al.[26], includes various performance-boosting features like the redundant computation skipping technique to speed up decoding and sparse matrix format for maintaining high efficiency with a large number of MACs in hardware. It also has a flexible hardware architecture to support diverse matrix multiplications and it ensures all intermediate computation values remain local, thereby eliminating the need for DRAM access and achieving rapid single-batch inference.

SpAtten[27], an efficient algorithm-architecture co-design, proposes token and head pruning and progressive quantization to reduce memory accesses and computations using a special engine to rank token and head importance scores.

ELSA[28], a hardware-software co-designed solution, utilizes an approximate self-attention algorithm to filter irrelevant query and key pairs and only performs exact computation for relevant pairs.

These emerging accelerators highlight a shift from general-purpose GPUs to specialized hardware, including TPUs. As Transformer models continue to evolve and grow in complexity, the development of such accelerators will play a crucial role in enabling real-time applications and managing the increasing computational requirements of these powerful models.

2.6 HW Platform

This thesis focuses on accelerating Transformer inference using the ITA accelerator[5] and ESP[6] [7] as hardware platforms.

ESP provides a flexible, open-source system-level design flow that supports the integration of accelerators like ITA into heterogeneous SoC architectures. It also enables the rapid prototyping of custom accelerators on FPGAs, which is essential for deploying machine learning models efficiently in embedded systems.

2.6.1 ITA: Integer Transformer Accelerator

The Integer Transformer Accelerator is a hardware accelerator designed to address the computational challenges posed by Transformer models and it is specifically optimized to accelerate the attention layer of the encoder portion. ITA adopts a weight stationary dataflow, minimizing data movement throughout the execution of the attention mechanism. Unlike traditional accelerators, it uses wide dot-product units, which enable deeper adder trees, enhancing efficiency by reducing the number of required operations.

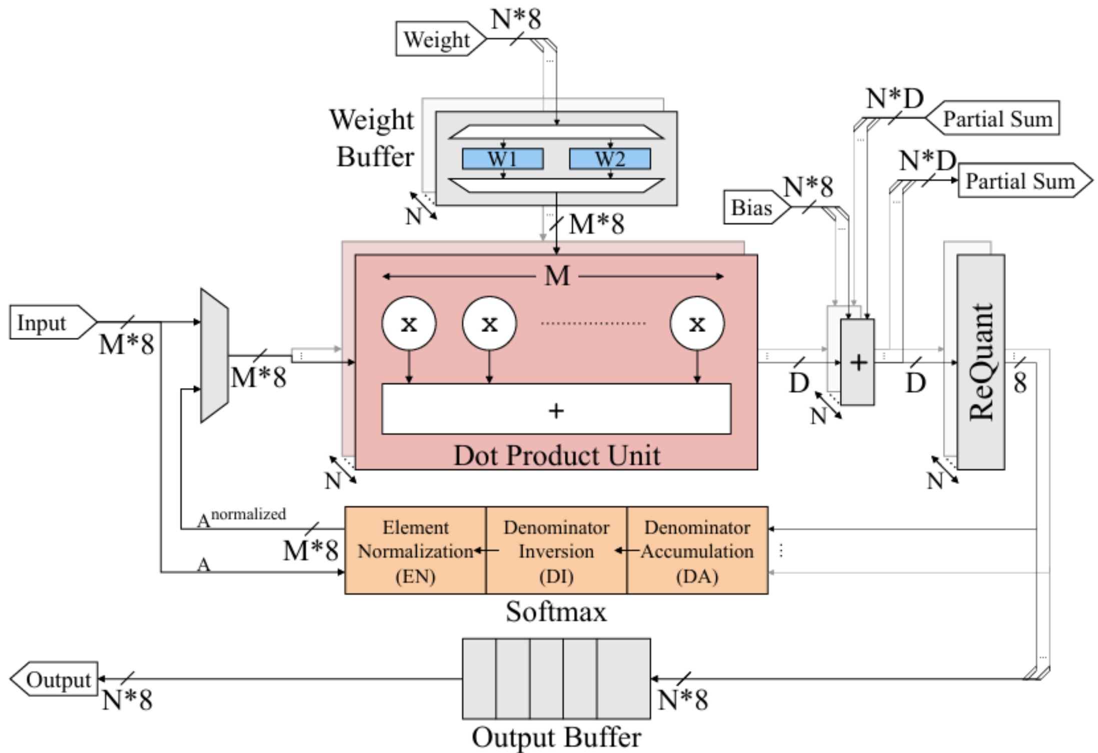

Figure 2.6: Architecture of ITA with 8-bit inputs and weights [5]

The key components are its handling of matrix multiplications involved in the self-attention mechanism and the softmax operation. ITA is designed to handle 8-bit integer quantized matrices. As you can see in Fig. 2.6, it includes N Processing Engines (PEs), each responsible for computing the dot product between two vectors of size M, and it works on tiles of size $M \times M$. Each PE uses 8-bit weights and activations, producing product results with higher precision of D-bit. The adders after PEs accumulate partial sums. Once outputs are fully accumulated, 8-bit biases are added to outputs, which are then re-quantized to 8-bit.

The softmax module works in two passes. In the first pass, it takes the elements of A, finds the maximum, and accumulates the denominator. In the second pass, when the attention matrix is supplied as input for the A × V computation, the softmax module normalizes them to probabilities before entering PEs. This streaming softmax implementation allows ITA to compute this non-linearity in an energy-efficient manner while minimizing data movement. The softmax values have a maximum value of 127 or -128 as "sumdotp" modules of the hardware can only do signed-signed operations.

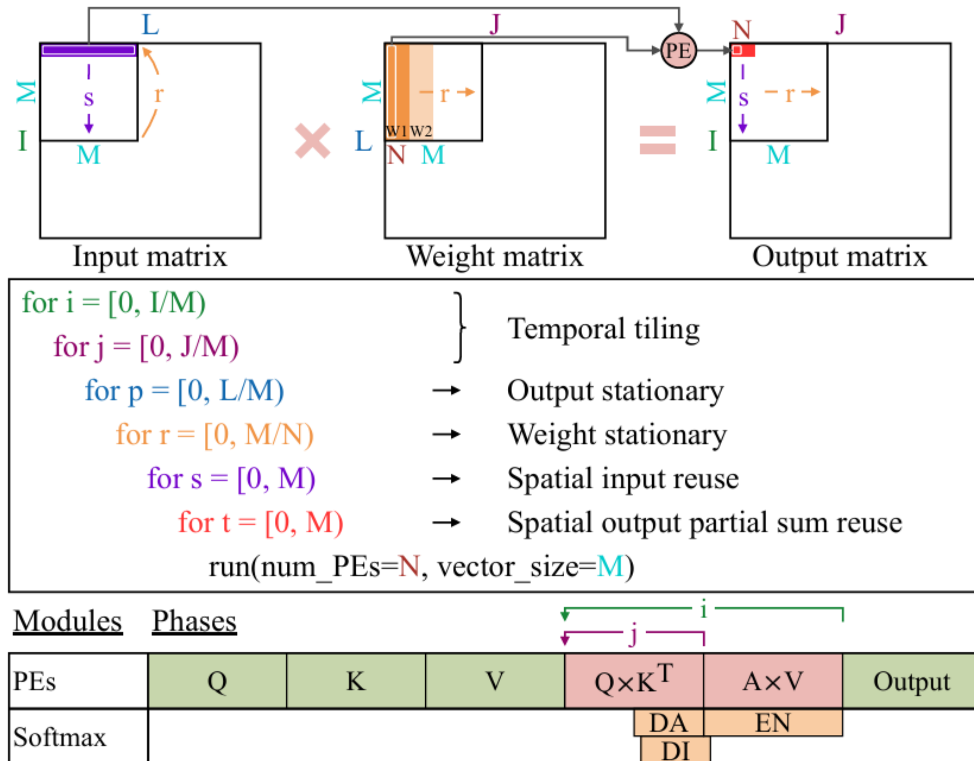

Figure 2.7: Tiling and computation phases [5]

The accelerator iterates over dimension L to achieve output stationarity in the outer loop, as shown in Fig. 2.7. Within each tile, ITA employs a weight-stationary regime and shares inputs among N PEs. Each PE operates on vectors of size M in the innermost loop. ITA computes linear layers sequentially but fuses Q $\times$ K^T and A $\times$ V in iterations of i. In each final iteration of a Q $\times$ K^T tile, the softmax module accumulates denominators partially (DA).

Once a row of M $\times$ J section of the attention matrix is completed, the softmax module inverts the denominator of the row (DI) and stores the inverted denominator. Then, M rows of A $\times$ V are computed while normalizing elements of A in the softmax module (EN). At the start of the next iteration of i, this unit is reset and the same steps are repeated until all iterations are completed.

ITA has been evaluated against several other state-of-the-art transformer accelerators, including OPTIMUS, SpAtten, and ELSA. In comparison, while other accelerators focus on exploiting sparsity or using floating-point arithmetic for softmax, ITA's integer-only approach leads to better performance in terms of energy consumption and area utilization.

HWPE interface

Hardware Processing Engines (HWPEs) [29] are special-purpose, memory-coupled accelerators that can be inserted in the SoC or cluster of a Parallel Ultra Low Power (PULP)[30] system to amplify its performance and energy efficiency in particular tasks. They interact directly with the shared memory architecture (like L1 Tightly Coupled Data Memory (TCDM) in PULP clusters). The HWPE interface[31], developed for the PULP platform facilitates the integration of accelerators. In this case, it helps the integration of the accelerator ITA into ESP.

As shown in Fig. 2.8, it provides three modules: a controller, one or multiple streamers, and the internal engine. The controller is the interface between the cores and the accelerator. It has a Finite State Machine (FSM) specific to the engine to govern the operation of the accelerator and a memory-mapped register file to keep parameters for the accelerator. The streamers are used to load and store data from the memories. Finally, the internal engine contains a hardware accelerator that accepts the streamer's data and the controller's configuration.

HWPE provides two types of streamers: one for input, source streamers, and one for output, sink streamers. The streamers utilize a simple valid-ready handshake protocol on the accelerator side, ensuring compatibility with most accelerators. Additionally, HWPE includes First-In-First-Out buffers (FIFOs), which can be instantiated and sized according to the specific needs of the accelerator.

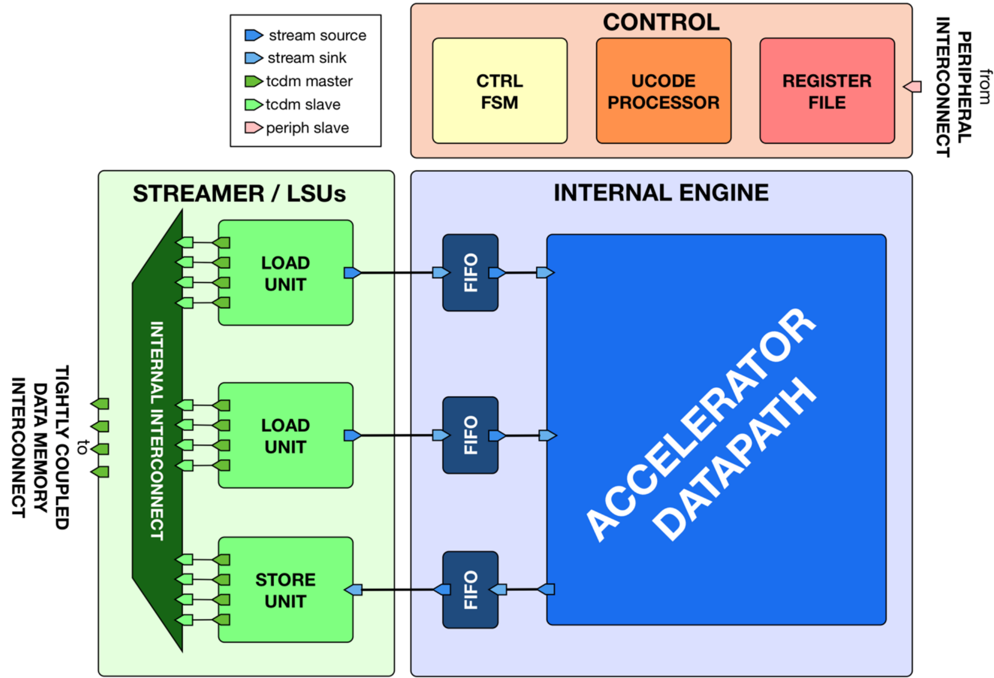

Figure 2.8: HWPE interface [29]

For ITA, N=16 dot product units are employed, each with a D=26-bit accumulator to support matrix dimensions up to 512 and a vector length of M=64. This configuration is chosen to fully utilize the memory-side bandwidth offered by the TCDM of the PULP system in which the accelerator had been originally integrated. ITA is designed with three input ports (input, weight, bias) and one output port. Since the four streamers are time-multiplexed, ITA requires 148B/cycle of maximum bandwidth to fetch the necessary data per cycle. As a result, 37 master ports on the TCDM interconnect are used for the HWPE subsystem.

ITA fetches up to two 64×64 8-bit inputs/weights, 64 24-bit bias values, and writes back 64×64 8-bit outputs to the memory.

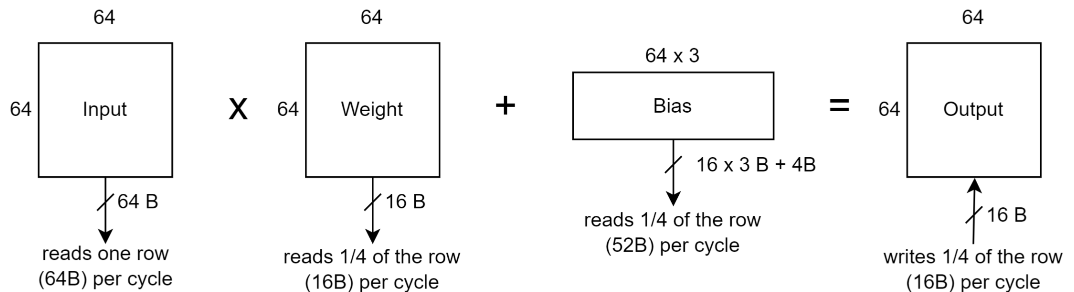

Figure 2.9: Data requirement

Every cycle, the necessary input data and weights are read to maintain full utilization of the processing units. The system uses streamers capable of generating up to three-dimensional loops, which ensure efficient access to memory locations and handle data transfers seamlessly. Partial sums are stored across iterations, and outputs are only written during the final cycle when the matrix multiplication is fully complete. Additionally, ITA requires input data and weights in every cycle to maintain 100% utilization.

In this version, the tiling feature of ITA is not available because the HWPE interface does not support it. To enable tiling, the interface would need to be modified accordingly.

2.6.2 ESP

Embedded Scalable Platforms is an open-source research platform designed for heterogeneous System-on-Chip design and programming. It is the result of years of research and teaching at Columbia University. The primary motivation behind ESP's development is the exponential growth of heterogeneous computing. These architectures are termed heterogeneous because they integrate general-purpose processors, such as CPUs, specialized processors with custom instruction sets, graphics processing units, tensor manipulation units, and highly specialized accelerators. This shift from homogeneous multi-core processors to heterogeneous SoCs is driven by the need for extremely energy-efficient computation.

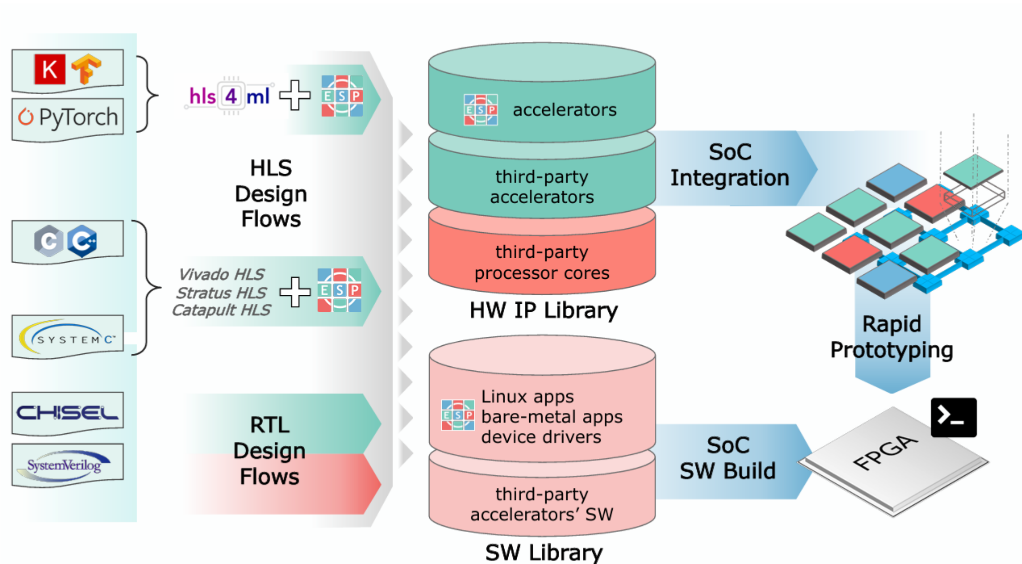

Figure 2.10: Design flows supported by ESP [7]

ESP provides a range of design and integration flows for SoC development. It simplifies the creation of complex and large SoCs, which can then be implemented on FPGAs for testing and prototyping. It supports the design and implementation of custom accelerators, as you can see in Fig. 2.10 through various supported design flows:

- C/C++ with Xilinx Vivado HLS
- C/C++ with Mentor Catapult HLS
- SystemC with Cadence Stratus
- Keras, PyTorch, ONNX, and TensorFlow with hls4ml
- Chisel, SystemVerilog, and VHDL for RTL design

Tiles form the foundation of the ESP architecture, with their number being selectable at design time based on application needs. Tiles may include:

- **Processor Tile:** Available cores include the 32-bit SPARC V8 Leon3, the 64-bit RISC-V CVA6-Ariane [32], and the 32-bit RISC-V IBEX. These cores come with private L1 caches and can boot Linux. A configurable private L2 cache can also be added. ESP supports system-level coherency and dedicated NoC planes for Input-Output (IO) and Interrupt Request (IRQ) channels.
- **Memory Tile:** Each memory tile connects the SoC to external memory. The hardware logic for handling memory address space is automatically created, with configurable memory sizes and coherent DMA units for data transfers.
- **Accelerator Tile:** Facilitates integration of accelerators for specific tasks. Accelerators interact with the memory hierarchy and can operate independently from processor cores. PLMs in accelerator tiles store local data and support various cache coherency models.
- **Auxiliary Tile:** Optional tile containing shared peripherals such as digital video interfaces, debug links, or monitor modules. This tile supports many devices but it is quite complex due to its flexibility.

ESP offers services for accelerator tile design, including dynamic voltage and frequency scaling, performance counters, and monitors. These services are set at design time, with reconfigurability options available at runtime.

A key feature of ESP stands in its hardware sockets. The accelerator socket allows for independent design of accelerators and their integration into the SoC [33]. Accelerators and third-party IP blocks, such as NVDLA, can be easily inserted into an accelerator tile without detailed configuration for virtual memory, DMA,

data transfers, and interrupt requests. The multi-plane NoC interacts solely with the socket, and third-party accelerators can have simplified sockets if they support only a subset of functionalities.

The ESP GUI facilitates SoC design by automatically discovering RTL codes of developed accelerators and enabling their insertion into tiles. Bare-metal test programs can be compiled, and the full RTL implementation of the SoC, including processor, memory, and I/O tiles, can be generated for FPGA prototyping.

RTL Design Flow

The RTL design flow in the Embedded Scalable Platform [34] facilitates the development of custom accelerators using Hardware Description Languages (HDLs) such as Verilog, VHDL, Chisel, or SystemVerilog. For instance, in this thesis, the ITA accelerator was developed using SystemVerilog, making it ideal to follow this flow for its seamless integration into ESP.

The design process began with a careful specification and design phase. The first step was to clearly define the functionality and performance requirements for the RTL accelerator. Once the requirements were established, the SystemVerilog RTL code was written to meet these specific needs, focusing on efficiency and scalability, and then, the next critical phase was the verification. This involved extensive testing using simulation tools to ensure that the RTL design met the functional and performance specifications laid out during the initial phase. Custom test-benches were created to simulate in various configurations, allowing for detailed observation of the accelerator's performance and correction of any identified issues early in the design process.

Following successful verification, the accelerator is integrated into the ESP system-on-chip (SoC) framework. Integration in ESP is particularly streamlined through the use of a modular socket-based architecture. It is connected to the platform via the ESP socket, which automatically handles communication with other system components such as the processor, memory tiles, and I/O peripherals. This step is crucial, as it ensured that the accelerator could interact efficiently with the rest of the SoC, leveraging ESP's scalable design to support different workloads and configurations.

With the integration complete, the design moved on to the synthesis and implementation phase. During synthesis, the RTL code was translated into a gate-level netlist, which was then mapped to the resources available on the target FPGA. This phase also involved ensuring that the design met the necessary timing constraints and resource utilization goals.

Finally, the RTL accelerator is prototyped on an FPGA, allowing for real-world testing within the context of the complete ESP system. This prototyping phase enabled the team to validate its performance in a full system, measuring its

execution speed, power consumption, and data throughput in relation to the rest of the SoC components. The FPGA prototype also provided a valuable opportunity to fine-tune the design based on real-world performance metrics.

ESP Socket Integration

The ESP socket is a crucial element in the integration of RTL-designed accelerators into the SoC. It abstracts the complex interfacing details. The ESP socket abstracts the interface between the accelerator and the rest of the SoC. It handles virtual memory, DMA, data transfers, and interrupt requests. This abstraction simplifies the design process by providing a standard interface for all accelerators.

The socket can be customized based on the specific needs of the accelerator. If a third-party accelerator or a new design has different interface requirements, the socket can be adjusted accordingly. For instance, the socket can be simplified if only a subset of functionalities is needed.

The multi-plane Network-on-Chip interacts with the ESP socket to manage communication between accelerators and other SoC components. This setup ensures efficient data transfer and communication within the system.

To guarantee successful integration with the ESP socket, it is important to follow specific guidelines when designing an accelerator for ESP using RTL [35]. Initially, it is crucial to establish the interfaces for the accelerator, such as data ports, control signals, and any unique requirements for DMA and interrupts, making sure that these interfaces meet the ESP socket specifications. Then, the accelerator should be connected to the NoC and other SoC components using the socket interface modules provided by ESP, which may require linking to typical interfaces for memory, I/O, and interrupt management.

Moreover, the socket settings need to be adjusted to align with the needs of the accelerator. This involves establishing memory-mapped registers and setting up DMA channels for optimized data transfer. At last, testing and validation are crucial stages, during which test-benches are created and executed to confirm the functionality of the accelerator inside the ESP socket. It is important to ensure that all interfaces operate correctly and that the accelerator performs as expected in the SoC context.

2.6.3 ESP accelerator specification

Fig. 2.11 illustrates the ESP accelerator socket and shows the three main sets of signals at the interface of an ESP accelerator: read and write port for data transfers through DMA requests, configuration port, and interrupt line.

The ESP accelerator follows a structured model for data processing and it comprises three main control blocks, which are the configuration, the load, and the

store, one or more computation blocks, and a customized PLM.

Once the accelerator's configuration registers are populated, the configuration block activates the rest of the components. The load block initiates a DMA read transaction to fetch input data from the system memory into the PLM. The computation blocks then process the available input data and generate corresponding outputs. Finally, the store block issues a DMA write request to send the processed data back to the system memory.

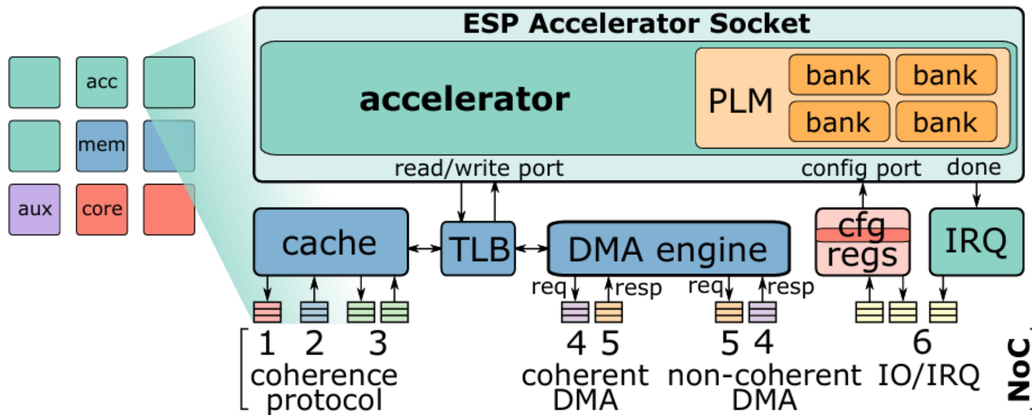

Figure 2.11: Block diagram of the ESP accelerator model [35]

The configuration block regulates the accelerator's execution by sampling both common and user-defined configuration registers, which are located in the ESP socket and mapped to memory. The configuration port uses different signals driven by the socket or the accelerator.

The clock signal (*clk*), provided by the ESP socket, synchronizes all operations within the accelerator. The reset signal (*rst*) is also driven by the socket and is used to initialize the accelerator, ensuring it is ready for a new task. This reset is active low and is asserted when the software clears the interrupt request to prepare the accelerator for the next invocation. If the accelerator needs to retain its state across multiple invocations, a custom reset mechanism can be implemented using a user-defined register. The configuration process is finished when the *conf_done* signal is asserted by the socket. This signal is active high and signals that the configuration registers are valid, triggering the start of the accelerator's computation. There can be up to 48 user-defined configuration registers, which provide custom parameters such as data dimensions or algorithm-specific control signals. These parameters are passed as inputs during the configuration phase and become valid when the *conf_done* signal is asserted.

Once the accelerator has completed its task, it raises the *acc_done* signal. This

active-high signal indicates that all data has been processed and the output has been written back to memory. Upon assertion of the *acc_done* signal, an interrupt request is generated and sent to the system's interrupt controller. The software then handles the interrupt, clearing the request and resetting the accelerator for the next execution. Additionally, the accelerator can output a 32-bit *debug* signal, which allows designers to monitor its internal state or report error codes. This debug information is accessible through the memory-mapped registers in the ESP socket.

Private Local Memory

Private local memories in ESP serve as the accelerator's working buffer, they operate as a temporary storage area for the accelerator's computations and are customized for its data path. PLMs store input data fetched from the system memory during DMA read transactions and hold output data before it is written back to memory via DMA write transactions.

They are generated using the ESP plmgen utility, which creates SRAM blocks based on the target technology. However, they are not memory-mapped, not exposed to software, and are not part of the SoC cache hierarchy. As a result, they have no external interface exposed to the ESP accelerator socket, and RTL designers are not required to comply with any hierarchy convention or signal-level protocol to implement a PLM.

DMA Transactions

Direct Memory Access is a core component of the ESP platform that allows accelerators to access system memory efficiently without CPU intervention. DMA is essential for moving data between the memory hierarchy and the accelerator's Private Local MemorieS, enabling high-performance computing by reducing the overhead on the CPU.

The master of the DMA, which is always an accelerator, initiates a DMA read transfer to load input data from the memory hierarchy into a PLM, and a DMA write transfer to store output data back to the memory. DMA transactions are initiated via the *dma_read_ctrl* and *dma_write_ctrl* control channels. These transactions are configured using three key parameters: the index, length, and size of the data transfer.

DMA control channels follow a simple protocol [36]: when both valid and ready control signals are set, the value of the data bus is sampled by the slave. From the accelerator viewpoint valid and ready are independent. The control valid signal is driven by the accelerator and it indicates a new DMA transaction request and when it is set all the data fields must be correct. The control ready signal, driven by the socket, indicates the ESP socket is ready to accept a new request.

Each DMA transaction begins by specifying the *index*, which defines the offset that is used to compute the starting address of the transaction. The *length* parameter determines the number of data beats to be transferred, while the *size* parameter indicates the width of the data token, which can vary between byte, word, or double-word, depending on the system's architecture. This signal is used to correct the NoC flits when the processor architecture follows the big-endian convention to store data in memory.

The DMA engine inside ESP translates virtual addresses into physical addresses using a page table, ensuring that the correct memory regions are accessed. Once the transfer is configured, data is transferred across the network-on-chip (NoC) between the accelerator and the memory. The transaction is synchronized using the ready and valid signals. The transfer completes when both signals are asserted in the same cycle, indicating that the data has been successfully received.

As shown in Fig.2.12, a read transaction begins with a single beat transfer through the DMA control channels. After the initial transfer is completed on the read control channel, the accelerator signals the socket to fetch the requested data by setting the ready signal high on the DMA read channel. Data is transferred successfully when both the ready and valid signals are high in the same clock cycle. The data transfer rate is flexible, allowing the accelerator to reduce the speed by lowering the ready signal if needed. However, the accelerator must eventually transfer the full number of beats specified in the request. Failing to complete the transfer can result in a deadlock within the socket, and in some cases, this deadlock can spread to the NoC and even to a memory tile.

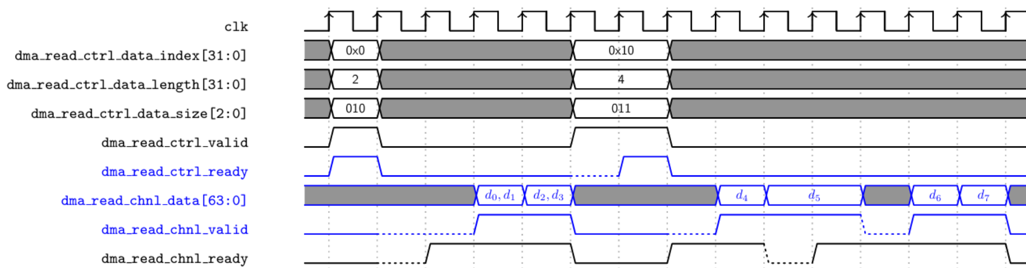

Figure 2.12: Example of a DMA read transaction in ESP [35]

For a DMA write transaction, as you can see in Fig.2.13 the processed output data is transferred back to the system memory through the *dma_write_ctrl* channel. The accelerator sends the data in beats, with a valid signal indicating when the data is ready for transfer. The transaction proceeds when both valid and ready signals are asserted simultaneously. Like the read transaction, the accelerator must transfer the exact number of beats specified by the length field. If the socket is not ready to accept data, the accelerator holds the data until the ready signal is

asserted.

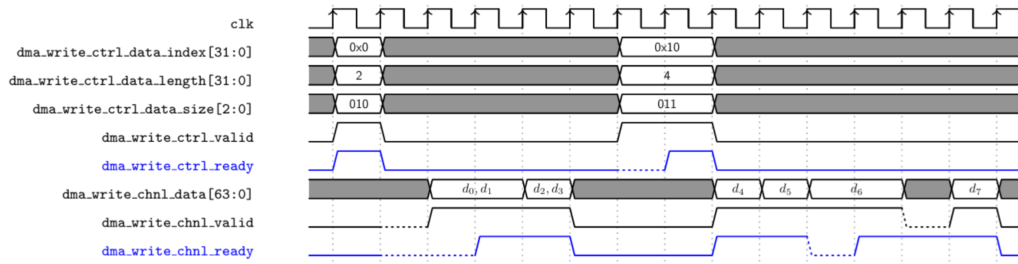

Figure 2.13: Example of a DMA write transaction in ESP [35]

Chapter 3

Accelerating Attention on ESP

Before delving deep into the integration between ITA and ESP, an understanding of the RTL accelerator code base, its pre-existing HWPE interface, and its test vector generator is needed. The repository consists of:

- `ModelSim` folder: Contains the Makefiles and scripts needed to simulate with ModelSim.

- `pyITA` folder: Includes the test generators for ITA. This folder contains all the Python scripts needed to obtain the files with the input data, the weights the biases, and the various intermediate and final outputs, which are located in another folder called "simvectors". The initial script is testGenerator.py, which is located outside the folder. This script is designed to generate test data for Multi-Head Attention layers. It can also export data in ONNX format.

- `sourcecode` folder: This is where the accelerator code, the interface and the various test-benches are kept.

After a thorough understanding of the accelerator's code base, we can move on to understand its interface and the various protocols that control it. As mentioned before, ITA, which is contained inside the internal engine (see Fig. 2.8), leverages a pre-existing interface. The accelerator interface reuses key components of the HWPE interface, such as streamers and parts of the control block, to manage the data flow.

3.1 Integration of one ITA instance into ESP

The first steps needed to integrate ITA onto ESP reside in memory-related connections, more precisely related to Private Local Memories and DMA transactions handling.

Firstly, we managed PLMs by introducing four dedicated instances, instead of relying on an L1 cache.

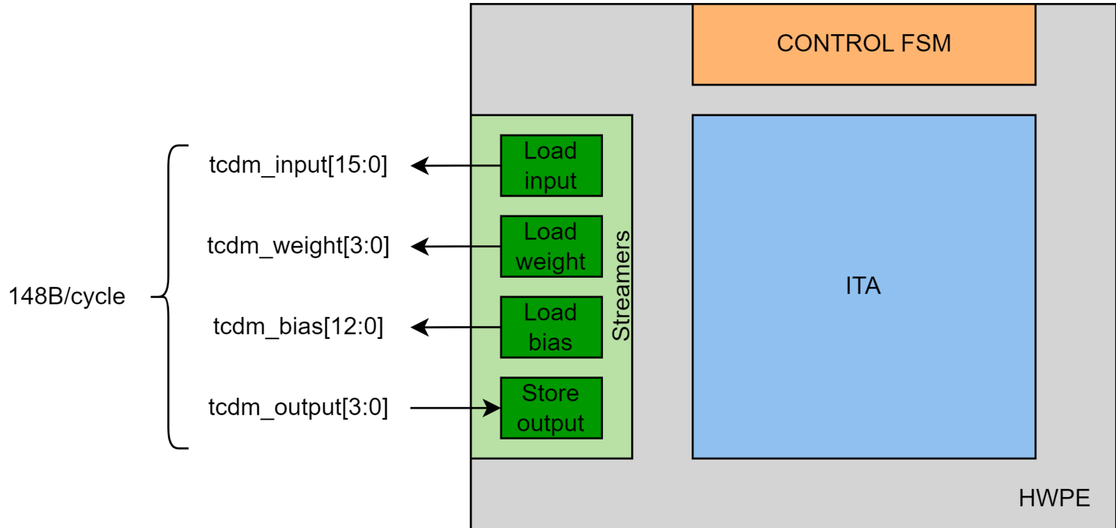

Figure 3.1: HWPE with ITA

Each streamer is paired with a separate PLM, which improves data management by reducing the latency associated with memory accesses. In terms of control logic, a portion of the pre-existing control unit is reused, while the register file that was part of the HWPE has been removed. The new design leverages the register file of the Ariane RISC-V processor to manage the configuration and control signals.

The next step regarded the DMA. A custom controller has been added to manage the DMA transactions efficiently. This controller ensures that data is transferred between the PLMs and the system memory without any unnecessary delays, optimizing throughput and enabling the accelerator to handle large volumes of data.

3.1.1 PLMs management

The generation of the four Private Local Memories for the accelerator was carried out using the Python script `plmgen.py`, which is part of the ESP framework. This script automates the creation of Verilog modules that define the memory blocks, customized for the specific accelerator needs. Each PLM is tailored to the memory

requirements of a different component in the accelerator, ensuring efficient data handling and access during computation.

Memory Configuration File: `memlist.txt`

The `memlist.txt` file defines the specifications for each of the four PLMs (input, weight, bias, and output memories). The configuration of each memory block is specified in terms of size, word width, and read/write ports. Below is the content of the `memlist.txt` file used for the PLM generation:

```
plm_INPUT 16384 32 16w:16r
plm_WEIGHT 16384 32 4w:4r
plm_BIAS 16384 32 16w:0r 0w:16r
plm_OUTPUT 16384 32 4w:4r
```

Each line in the file specifies:

- `Memory Name`: Identifies the PLM, such as `plm_INPUT` for input data memory.
- `Words`: The number of logic words in memory
- `Word Width`: The bit width of each memory word (32 bits).
- `Ports Configuration`: The number of write and read ports, specified as `<write ports> w : <read ports> r`.

The `plmgen.py` script reads the `memlist.txt` configuration file to generate the Verilog files for each PLM. The generated files include all the logic required to implement the memories in hardware.

By using this automated tool, the integration process of PLMs into the accelerator design is greatly simplified, avoiding the need to manually design memory modules for each specific configuration.

Generated PLM Example: `plm_INPUT.v`

The `plm_INPUT.v` file represents the input data memory of the accelerator. This PLM is responsible for storing input data fetched by the streamers, ensuring efficient access during computation. The following are the key input and output signals used in this PLM:

- `CLK`: The clock signal used to synchronize the read and write operations across the PLM.

- `RST_N`: An active-low reset signal that initializes the memory. When asserted low, this signal resets the internal state of the memory. This ensures that all previous data is cleared, and the memory is ready for new operations.

- `WE[15:0]`: A 16-bit signal used to selectively enable specific write operations across multiple memory ports. Each bit corresponds to one port, meaning that the signal can control up to 16 different ports individually. For instance, if `WE[0]` is set to high, the data present on the `D` signal will be written to the memory using port 0. If `WE[1]` is high, port 1 is enabled for writing, and so on. This write enable allows the system to write data only to the specific locations required.

- `WEM[15:0][31:0]`: The Write Enable Mask is a vector of 16 32-bit signals that provides finer control over memory writes. Each `WEM[i][j]` bit corresponds to the j-th bit of the i-th port being written to the memory. This allows the system to selectively enable or disable writes at the bit-level, ensuring that only specific bits are updated, while the rest remain unchanged. For example, if `WEM[0][7]` is high, only the 8th bit of the data word associated with port 0 will be written, while the other bits remain unchanged. This mechanism enhances flexibility.

- `A[31:0][13:0]`: A vector of 32 14-bit address signals that specify the memory location for both read and write ports. This address points to the location in memory where the data is either stored or retrieved. The address bus points to the specific location in memory where the data will be written or from which it will be read. The 14-bit width allows for addressing a total of $2^{14} = 16{,}384$ memory locations.

- `D[15:0][31:0]`: A vector of 16 32-bit input data signals carrying the data to be written to the memory at the address specified by `A`, controlled by `CE`, `WE` and `WEM`.

- `CE[31:0]`: A 32-bits signal that activates the PLM ports during the read and write operations, where each bit manages one port. When one of this bit is high, the corresponding PLM port is active and can perform read or write tasks depending on the other control signals. If `CE[0]` is not asserted for example, the memory port 0 will not respond to any operations, in this case to a write operation.

- `Q[15:0][31:0]`: A vector of 16 32-bit output data signals that provide the data read from the memory location specified by the `A` signal. This signal is driven when the PLM is enabled for a read operation.

The architecture of each PLM is designed to meet the needs of high-throughput accelerators. In this case, the input data PLM is designed to handle 16 read and write operations, since it has 32 ports, allowing the accelerator to fetch and store data efficiently during each computation cycle. This architecture supports high levels of parallelism when accessing memory, as it not only writes the intermediate outputs back to memory after each computation phase but also simultaneously reads the data required for the current phase. Some of the intermediate output are stored also in the PLM dedicated to weights, as will be described in later sections.

This is particularly important for accelerators targeting data-intensive tasks like Transformer models, where multiple inputs and outputs need to be handled simultaneously.

The other PLMs (weight, bias, and output) follow a similar structure, with slight variations in the number of read/write ports, as defined in the `memlist.txt` file. These differences ensure that each PLM is optimized for the specific type of data it handles.

For example, since in the PLM for the weights we have just 8 ports (4 for reading and 4 for writing), the signal CE will be 8-bit to control each port with one bit and signal A will be an array of 8 14-bit signals. Therefore, each signal dimension will correspond with the number of ports on the PLM.

Integration of PLMs with ITA Streamer

The integration of the four PLMs into ITA is achieved through custom wrappers, with each PLM having a dedicated wrapper to manage data flow between the accelerator, its designated streamer, and its dedicated memory. These additional wrappers interact directly with the ITA streamers, which handle the data movement between the computation units and the PLMs. In previous hardware configurations, such as those used in PULP systems, the streamers interfaced with L1 memory through TCDM (Tightly-Coupled Data Memory) interfaces. In this design, the TCDM interface is reused, but instead of an L1 cache, each PLM is directly mapped to a specific streamer, ensuring faster access and efficient data handling.

The primary role of these wrappers is to manage read and write operations, synchronize data flow, and ensure that memory accesses are performed correctly based on the input signals from the ITA streamer. Below is a description of each wrapper and its functionality:

- **PLM Weight Wrapper:** The `plm_weight_wrapper` is responsible for managing the weight data. It has four memory ports for reading and four for writing (`MP=4`), corresponding to the TCDM ports from the ITA streamer. The TCDM interface provides the `req`, `gnt`, `add`, `wen`, `be`, `data`, and `r_data` signals. These signals manage the request, grant, address, write enable, byte enable, and data transfer operations, respectively.

This PLM handles also intermediate data storage during different computation phases. Through the use of external enable signals controlled by the DMA controller, which will be discussed later in the thesis, the intermediate output data is written back into the PLM, ensuring that the new required data are correctly synchronized with the ongoing computations.

- **PLM Input Wrapper:** The `plm_input_wrapper` manages the activations for the accelerator. It is configured with 16 memory ports (`MP=16`) to handle higher throughput. The TCDM interface signals are mapped similarly to the weight wrapper. This wrapper ensures that data is available for computation at every phase. As said before, this PLM handles intermediate output data storage.

- **PLM Bias Wrapper:** The `plm_bias_wrapper` is responsible for storing bias values. In this case, `MP=13`, corresponds to the number of TCDM channels required for reading bias data. However, due to constraints in the number of ports that could be generated, 16 ports for each operation (`MP=16`) were used. The `plmgen.py` tool only supports power-of-two values for the number of write ports, limiting flexibility in port configuration.

- **PLM Output Wrapper:** The `plm_output_wrapper` handles the output data produced by the accelerator. It is configured with four ports for reading and four for writing, (`MP=4`), which match the number of TCDM ports in the ITA streamer for output data. This wrapper is responsible for writing the processed output data back into memory and ensuring that the data is available for subsequent processing or storage.

The TCDM interface used in each wrapper provides several key signals to manage memory operations. These include:

- `req`: A request signal that initiates a memory operation.

- `gnt`: A grant signal that indicates when a memory operation can proceed.

- `add`: A memory address signal that specifies the location in memory for a read or write operation.

- `wen`: A write enable signal that controls when data is written to memory.

- `be`: A byte enable signal that specifies which bytes within the memory word are being written.

- `data`: The data being written to or read from memory.

- `r_data`: The data read from memory.

- `r_valid`: A signal indicating that valid data is available after a read operation.

The number of ports in each wrapper is determined by the number of TCDM channels required by the ITA streamer. By replacing the L1 cache memory with four distinct PLMs, the design eliminates the need for complex cache management and reduces latency, because every PLM is always ready for read and write operations. This is the reason why tcdm_gnt is just set to '1'.

3.1.2 DMA Controller Design and Implementation

The DMA (Direct Memory Access) controller plays a critical role in orchestrating data movement between the main memory and the Private Local Memories of the accelerator. This section describes the design and implementation of the DMA controller, developed to meet the specific requirements of the ESP accelerator specification. The DMA controller is responsible for handling read and write transactions to the PLMs, managing intermediate results, and ensuring efficient data transfer during the various computation phases.

The DMA controller operates through a sequence of well-defined phases, each managing a specific aspect of the data flow within the accelerator. The primary ones are the following:

- `Data Loading Phase:` Here, the required data (input, weights, and bias) is fetched from the main memory and loaded into the corresponding PLM. This phase uses the ESP DMA transaction protocols to handle memory reads from the external memory and transfers data into the PLMs for use during the computation.

- `Data Write-back Phase:` When the accelerator computation is completed, the final output data is written back from the output PLM to the main memory. The DMA controller performs this operation by transferring the results from the output PLM to the main memory using DMA write transactions.

The DMA controller is structured around several distinct states that manage the flow of operations from initialization to completion. These states are: `Idle`, `Input`, `Weight`, `Bias`, `Intermediate`, `Output`, and `Done`.

Initially, it resides in the `Idle` state, where the accelerator awaits the configuration registers to be set. Once the socket sends the `conf_done` signal, indicating that all configurations have been correctly initialized, it transitions into the `Loading Phase`. During this phase, the accelerator begins loading the necessary data for computation. Upon completing the data loading, the controller moves into the `Intermediate` state, where the main computational tasks are carried out. When the accelerator finishes the computation, it raises the `evt_o` signal and the controller

enters the `Output` state. In this phase, the accelerator writes the results back to the main memory via the DMA. Once all expected data transfers are completed, the controller transitions to the `Done` state. At this point, the `acc_done` signal is raised, notifying the socket that the accelerator has finished its operations. Finally, the controller returns to the `Idle` state, awaiting the next configuration or operation to begin.

Now, let's take a closer look at what occurs during each state.

Data Loading Phase

During the data loading phase, after the Idle state, the DMA controller is responsible for reading the input data, weights, and bias values from the main memory and transferring them into the respective PLMs (`plm_input`, `plm_weight`, `plm_bias`). This phase involves several distinct states in the controller, each responsible for handling specific types of data. The following steps occur:

- In the `Input State`, the DMA controller initiates three separate read transactions, one for each input (q, k, and v). For each transaction, the base address from which data needs to be read is updated using the configuration registers. The signals `dma_read_ctrl_valid`, `dma_read_ctrl_data_index`, `dma_read_ctrl_data_length`, and `dma_read_ctrl_data_size` are set accordingly to initiate the transaction. For example:

  ```
  dma_read_ctrl_data_length = 32'h200;
  dma_read_ctrl_data_size = 3'b011;
  dma_read_ctrl_valid = 1'b1;
  dma_read_ctrl_data_index = address;
  ```

  The values for `dma_read_ctrl_data_length` and `dma_read_ctrl_data_size` are chosen based on the data transfer requirements and the architecture of the Ariane processor. Referring to Tab.3.1, we are working with double words (DWORD), which correspond to 64 bits (8 bytes) per transfer. This is essential because the Ariane processor handles data in 64-bit, which matches the size of the DWORD. Each input has dimensions of $S \times E$, so 64×64 bytes, totaling 4096 bytes. Since we are transferring data in 8 bytes, the number of transactions required is:

  $$\frac{64 \times 64}{8} = 512$$

  In hexadecimal, 512 bytes is equivalent to 0x200, which explains why `dma_read_ctrl_data_length` is set to 32'h200. Similarly, the value for `dma_read_ctrl_data_size` is set to 3'b011.

Encoding	Name	Bitwidth
000	BYTE	8
001	HWORD	16
010	WORD	32
011	DWORD	64

Table 3.1: Encoding of DMA size

Then, the address is updated after each DMA transaction to point to the next portion of data to be read. The controller transitions out of this state once all input data has been successfully transferred.

- In the `Weight State`, four separate read transactions are initiated, one for each weight matrix tile. The base address for the weight data is similarly updated after each transaction. This process ensures that all weights required for computation are loaded into `plm_weight`. Once the four sets of weights are fully transferred, the controller moves to the next state.
- The `Bias State` operates similarly to the previous states and it handles four separate transactions for the bias values. The controller loads bias data into `plm_bias` by setting the same control signals, with updated addresses for each transaction. The controller transitions to the *Intermediate* state once all the bias values are loaded.

The transitions between these states are controlled by internal counters, which track how much data has been loaded. When the required data has been transferred, the controller proceeds to the next state. The number of DMA protocol calls is determined by the data size and memory bandwidth, with each call transferring a fixed amount of data per cycle.

Intermediate Phase

The intermediate phase is where the computation of the ITA accelerator takes place. To effectively manage the execution flow, an additional `Intermediate State` was introduced, which monitors the overall execution of the accelerator and ensures that computations are synchronized with the PLM read/write operations.

During this phase, the signal `evt_o` is used to determine when the accelerator has finished processing a particular operation. This phase is critical for enabling or disabling read and write accesses to the PLMs. For instance:

- The **Q (Query) sub-phase** begins with reading data from `plm_input` and writing intermediate results back to the PLM. The signal `write_en_input` is activated during this process.

- In the **K (Key) and V (Value) sub-phases**, weights and biases are read from `plm_weight` and `plm_bias`, respectively. Intermediate results are written back to the weight PLM, and the enable signal `write_en_weight` controls this operation.

- The **QK (Query-Key Multiplication sub-phase)** the intermediate results are written back to the PLM for inputs (`plm_input`), with the signal `write_en_input` activated during this process. Following this, also during the **AV (Attention-Value sub-phase)**, the results written back into `plm_input`.

- The **OW (Output Write sub-phase)** retrieves the final intermediate results and prepares them for the output PLM, with the signal `write_en_output` managing the write process.

This phase ensures the proper sequencing of operations through the use of control signals, which were kept within the controller to maintain synchronization between the PLMs and the DMA. Without this intermediate phase, it would be difficult to manage the complex dependencies between the local memories and the the accelerator.

Data Write-back Phase

Once the intermediate computations are complete, the DMA controller initiates the final phase, the `Output State`. Here, the output data stored in `plm_output` is written back to the main memory through DMA write transactions. The controller uses the signals `dma_write_ctrl_valid`, `dma_write_ctrl_data_index`, and `dma_write_chnl_valid` to initiate the write-back process. Each DMA transaction transfers a portion of the output data, similar to the loading phase. The controller transitions to the `Done State` once all the output data has been written to memory.

In the `Done State`, the controller sets the signal `acc_done` to 1, indicating to the socket that the accelerator has completed its execution. The system can now transition back to the `Idle State`, ready for the next operation.

3.1.3 Accelerator Integration

Now the integration process of ITA into the ESP platform will be discussed. Initially, all remaining unnecessary components, such as the register file that came with the interface, must be removed. Then, the process proceeds with the creation of the accelerator tile socket using ESP's tools. This integration allows ITA to communicate with the system's memory using DMA transactions, with the Ariane processor, and to interact with the ESP framework.

As a first step, the register file, that was inside the ITA interface, was removed, since ITA is supposed to utilize the register file of the ESP platform. This modification simplifies the design of the accelerator by reducing hardware redundancy and enables more efficient integration within the ESP infrastructure. Following this, ITA control logic had to be adjusted to work seamlessly with the new register file.

After this removal, the accelerator skeleton must be generated. For this purpose, ESP provides an interactive script that automatically does this, along with its software test applications and device driver. This script, `accgen.sh`, generates an empty Verilog/SystemVerilog top-level entities. These are the base structures for the accelerator, with the correct interface to allow automated integration within ESP. Even though the skeleton is empty initially, the registers and interfaces are defined by the user, to meet the requirements for DMA transactions, configuration settings, and connections to the ESP NoC. The generated folders include:

- `hw/`: Contains the accelerator hardware files and skeleton code for the RTL implementation.

- `sw/`: Contains software test applications for both bare-metal and Linux environments, used to test the functionality of the accelerator.

ESP platform automatically generates two interface files for the accelerator. These include top-level entities for both 32-bit and 64-bit architectures, named in this case `ita_rtl_basic_dma32` and `ita_rtl_basic_dma64`, respectively. For this integration, the 64-bit version was selected because the Ariane RISC-V processor operates on 64-bit data.

The `ita_rtl_basic_dma64.sv` file represents the top entity of the accelerator, which interfaces with the DMA control system, the ITA accelerator, and all the plm wrappers, as depicted in Fig. 3.2

The top-level module has several configuration inputs for several memory pointers, such as input, weight, and bias pointers, as well as multiple configuration parameters, used for the requantization, and tile information. These configuration inputs allow for flexibility in the computation. The module also interfaces with DMA channels, allowing for both read and write operations and uses DMA control signals to coordinate data transactions, as described before.

The file also defines several parameters related to memory partitioning. To manage these partitions, the design defines multiple signals to handle memory grant, read valid, and data transfer across the memory system for each data type (input, weight, bias, and output). These signals are then interfaced with the DMA and other components such as `hci_core_intf` and `hwpe_stream_intf_tcdm`. A set of configuration registers, defined as an array `conf_info_regs`, stores all the configuration parameters.

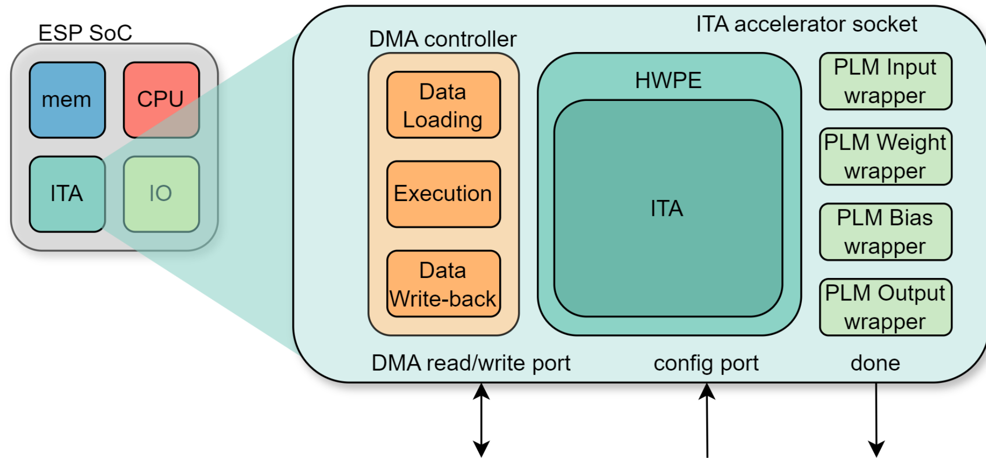

Figure 3.2: Block diagram of ITA accelerator tile in a 2×2 ESP SoC

Now everything is ready to set up the entire environment and to prepare the bare-metal application.

Preparing for Bare-Metal Application Development

At this stage, after filling the `hw/` folder with all the RTL files to simulate the environment, the next step is to proceed with the development of the bare-metal application. A bare-metal application is a software program that runs directly on the hardware without the support of an operating system, providing full control over the hardware resources. The bare-metal application is responsible for managing the execution of the ITA accelerator.

The application starts with the initialization of various parameters, such as sequence length, embedding size, projection space, and the number of heads, which dictate the size of input matrices, weights, biases, and outputs. A unique accelerator ID is passed as three hexadecimal digits, which in this case is `0x053` (selected when running the `accgen.sh` script). Additionally, the accelerator name, `"sld,ita_rtl"`, and the size of contiguous chunks for scatter/gather are also defined.

```
1  #define SLD_ITA 0x053
2  #define DEV_NAME "sld,ita_rtl"
3
4  #define SEQUENCE_LEN 64
5  #define PROJECTION_SPACE 64
6  #define EMBEDDING_SIZE 64
7  #define N_HEADS 1
8
9  /* Size of the contiguous chunks for scatter/gather */
10 #define CHUNK_SHIFT 20
```

```
11 #define CHUNK_SIZE BIT(CHUNK_SHIFT)
12 #define NCHUNK(_sz) ((_sz % CHUNK_SIZE == 0) ? \
13                 (_sz / CHUNK_SIZE) :      \
14                 (_sz / CHUNK_SIZE) + 1)
```

Listing 3.1: Parameter Definitions for Accelerator

Then, user-defined registers are established through `#define` directives. These registers correspond to specific memory-mapped locations that will be used to control and pass data to the hardware accelerator. For instance,

#define ITA_BIAS_PTR1_REG 0xa8

is a directive that defines the symbolic name ITA_BIAS_PTR1_REG for the hardware register located at memory address 0xa8. This address corresponds to a specific function within the accelerator, such as handling bias data in this case.

Based on the memory map of the data, the pointers, or base offsets, have been calculated incrementally to ensure correct alignment of the data blocks, as you can see from the Tab. 3.2.

Table 3.2: Memory Map and base pointers

Memory Map	Base Pointer	Dimension Formula
q (S x E)	`base_ptr0`	0
k (S x E)	`base_ptr1`	`base_ptr0 + SEQ_LEN * EMB_SIZE`
Wq (E x P)	`base_ptr2`	`base_ptr1 + SEQ_LEN * EMB_SIZE`
Wk (E x P)	`base_ptr3`	`base_ptr2 + PROJ_SPACE * EMB_SIZE`
Wv (E x P)	`base_ptr4`	`base_ptr3 + PROJ_SPACE * EMB_SIZE`
Wo (P x E)	`base_ptr5`	`base_ptr4 + PROJ_SPACE * EMB_SIZE`
Bq (P x 3)	`base_ptr6`	`base_ptr5 + PROJ_SPACE * EMB_SIZE`
Bk (P x 3)	`base_ptr7`	`base_ptr6 + PROJ_SPACE * 3`
Bv (P x 3)	`base_ptr8`	`base_ptr7 + PROJ_SPACE * 3`
Bo (E x 3)	`base_ptr9`	`base_ptr8 + PROJ_SPACE * 3`
Q (S x P)	`base_ptr10`	`base_ptr9 + base_ptr1 + EMB_SIZE * 3`
K (S x P)	`base_ptr11`	`base_ptr10 + SEQ_LEN * PROJ_SPACE`
V (S x P)	`base_ptr12`	`base_ptr11 + SEQ_LEN * PROJ_SPACE`
QK (S x S)	`base_ptr13`	`base_ptr12 + SEQ_LEN * PROJ_SPACE`
AV (S x P)	`base_ptr14`	`base_ptr13 + SEQ_LEN * SEQ_LEN`
OW (S x E)	`base_ptr15`	`base_ptr14 + SEQ_LEN * PROJ_SPACE`

Each pointer is calculated by adding the required memory space for the preceding data structure, ensuring that all data (inputs, weights, biases) is correctly positioned

in memory. Using the accelerator identifiers, the probe function can be utilized to verify if the device has been properly included in the SoC.

$$\text{ndev} = \text{probe}(\&\text{espdevs1}, \text{VENDOR_SLD}, \text{SLD_ITA}, \text{DEV_NAME})$$

The espdevs structure also stores valuable details about the location of the device under test (most functions, structures, and constants used by the bare-metal application are defined in the esp_accelerator.h and esp_probe.h headers files).

Then, the core allocates memory to hold the input and output, and reference data (gold) and the corresponding page table is populated.

```
mem_size = N_HEADS * ( (SEQUENCE_LEN*EMBEDDING_SIZE*2 +
    EMBEDDING_SIZE*PROJECTION_SPACE*4 + PROJECTION_SPACE*3*3 +
    EMBEDDING_SIZE*3) + (SEQUENCE_LEN*EMBEDDING_SIZE) );
gold_size = N_HEADS * SEQUENCE_LEN * EMBEDDING_SIZE;
// Allocate memory
gold = aligned_malloc(gold_size);
mem = aligned_malloc(mem_size);
// Allocate and populate page table
ptable = aligned_malloc(NCHUNK(tot_size) * sizeof(unsigned *));
for (i = 0; i < NCHUNK(tot_size); i++){
    ptable[i] = (unsigned *) &mem[i * (CHUNK_SIZE / sizeof(token_t))
    ];
}
```

Listing 3.2: Memory allocation and page table setup

Additionally, header files have been used to include test data. These files were modified to fit the 64-bit structure of the implementation. Further details on these modifications will be discussed in the following section.

```
#include "mem.h"
#include "gold.h"
#include "rqs_mul.h"
#include "rqs_add.h"
#include "rqs_shift.h"
```

Listing 3.3: Header file inclusions

At this point, the processor checks whether the DMA and the Translation Look-aside Buffer (TLB) are enabled or not, and all the common and accelerator-specific configuration parameters are written in the registers. The input and output offset registers, in this application, are set to zero because the input and output data are allocated at the default offsets with respect to a virtual memory region reserved for the accelerator. Once the configuration is complete, the accelerator is started, and the system waits for its completion by polling the status register.

```
    // Use the following if input and output data are not allocated
    at the default offsets
```

```
2     iowrite32(&dev, SRC_OFFSET_REG, 0);
3     iowrite32(&dev, DST_OFFSET_REG, 0);
4     // Start accelerators
5     iowrite32(&dev, CMD_REG, CMD_MASK_START);
6
7     // Wait for completion
8     done = 0;
9     while (!done) {
10        done = ioread32(&dev, STATUS_REG);
11        done &= STATUS_MASK_DONE;
12    }
13    iowrite32(&dev, CMD_REG, 0x0);
```

Listing 3.4: Starting and monitoring accelerator

Once processing is finished, the output is validated through a comparison with a pre-calculated gold standard.

Memory Layout and Data Preparation

To ensure that the input data is correctly aligned with the 64-bit architecture of the Ariane processor, the data was processed through a Python script that merged 32-bit data lines of the `mem.txt` file and the `Output.txt` file, generated from the ITA test generator, into a header files with 64-bit data lines. This process reduced the total number of lines by half, ensuring compatibility with the system. Below is the Python script used to generate the memory configuration files:

```
1  with open('mem.txt', 'r') as fileA:
2      lines = fileA.readlines()
3  fileA.close()
4  with open('Output.txt', 'r') as fileB:
5      lines_out = fileB.readlines()
6  fileB.close()
7
8  i = -1
9  j = 0
10 with open('mem.h','w') as header_file:
11     for l in lines:
12         i+=1
13         if i%2==0:
14             header_file.write("mem[%d]" % (j))
15             header_file.write(" = (unsigned long long) 0x%s" % (l.
    rstrip('\n')))
16             j+=1
17         else:
18             header_file.write("%s" % (l.rstrip('\n')))
19             header_file.write(";\n")
20 header_file.close()
```

```
21
22 i = -1
23 j = 0
24 with open('gold.h','w') as header_file:
25     for l in lines_out:
26         i+=1
27         if i%2==0:
28             header_file.write("gold[%d]" % (j))
29             header_file.write(" = (unsigned long long) 0x%s" % (l.
    rstrip('\n')))
30             j+=1
31         else:
32             header_file.write("%s" % (l.rstrip('\n')))
33             header_file.write(";\n")
34 header_file.close()
```

Listing 3.5: Data header creator file

This script reads the 32-bit data lines from the files, combines two consecutive lines into a single 64-bit value, and writes the result to a header file. Similarly, the output data is processed and stored in a separate header file for validation purposes. Due to this configuration, two ports were utilized simultaneously when writing input data received from the DMA into the PLM. This approach accelerated the data transfer by allowing two pieces of data to be written concurrently. Similarly, when writing the final outputs, two 32-bit data values are read and concatenated to form a single 64-bit value, aligning with the DMA's requirements.

In addition to this Python script to generate the header file for input and golden output, three other Python files were written for the other required data.

3.1.4 SoC Generation and Simulation

After RTL code management and the bare-metal application development, the next step stands in the SoC generation. Firstly, the used directory is the board one, in this case the Virtex UltraScale+ FPGA VCU118 (xilinx-vcu118-xcvu9p) folder. This is the FPGA board targeted for the SoC.

After the creation of the RTL accelerator folder (ita_rtl for this project), ESP automatically discovers it in the library of components and generates a set of make targets for it. Installing the accelerator in the FPGA tech folder can be done by running the `make ita_rtl-hls` command. After the installation, the accelerator can be instantiated in the SoC with the ESP configuration GUI, by executing the `make esp-xconfig` command. Following the grid dimension must be chosen, as in Fig.3.3, to accommodate ITA, the memory tile, and the 64-bit Ariane processor. Initially, the integration is achieved with only an instance of the accelerator, therefore a 2x2 matrix is enough.

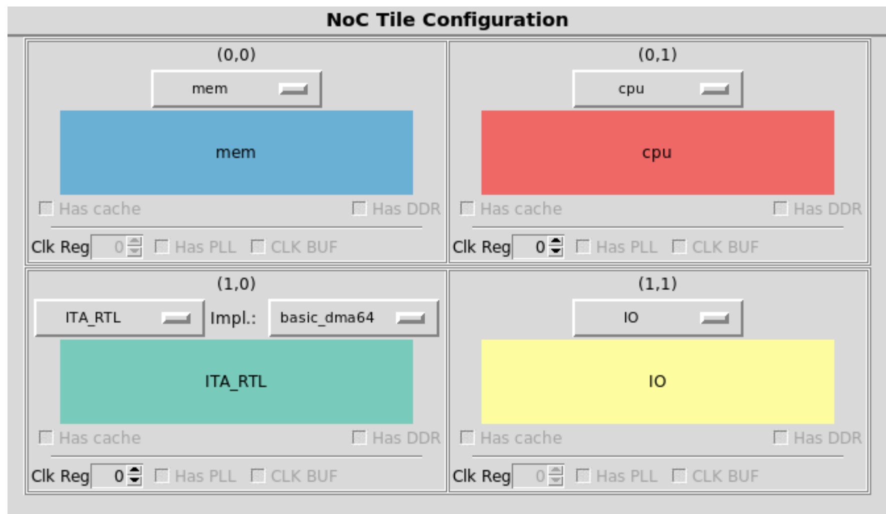

Figure 3.3: ESP GUI with one ITA accelerator

The host machine communicates with the ESP SoC through an Ethernet debug interface. As shown below in Fig. 3.4, the static IP of the debug interface can be configured by editing the IP and MAC addresses in the Debug Link section of the ESP GUI. It's possible to set them based on the FPGA board you want to use later.

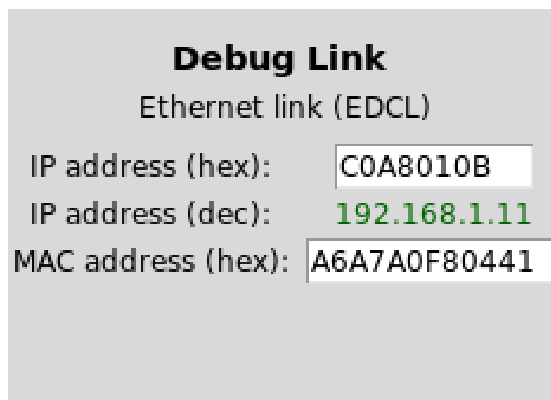

Figure 3.4: Debug Link of the ESP GUI

With the NoC Tile Configuration set, the overall configuration of the SoC can be generated in a user-friendly way powered by ESP, to finally build the SoC.

Given the built SoC, software such as the previously discussed bare-metal application can be compiled. For instance, for the bare-metal application the command `make ita_rtl-baremetal` is run. With both the HW and SW ready the simulation process can be started. With ModelSim, the simulation can be run

by executing `TEST_PROGRAM=./soft-build/ariane/baremetal/ita.exe make sim[-gui]`.

For a more comprehensive view, the latter command can be enhanced with the *-gui* parameter, which simplifies substantially the debugging process, otherwise it has to be carried out through prints that signal whether the final result of the simulation is correct or not. If at the end of the program, there have been found zero errors and the message `Failure: Program Completed!`, the final outputs calculated by the accelerator correspond to the golden outputs.

3.1.5 FPGA Deployment

Finally, the SoC design can be deployed on the FPGA board. This requires the generation of the corresponding bitstream, which can be achieved using Xilinx Vivado 2023.2. A makefile for this task is already available in the target's FPGA folder, it is then possible to run the `make vivado-syn` command. This process is very heavy and long, and at the end of it in the board's folder, the bitstream called `top.bit` is generated.

Once the bitstream file is ready, the correct FPGA board must be selected for usage in the website `https://espdev.cs.columbia.edu/fpgaboards.htm` .

During this thesis, the selected board is the Virtex UltraScale+ VCU118 (01). Additionally, also the Vivado version, used to generate the bitstream, has to be selected, and finally the application requires the IP address of the machine. After this process, ESP connects to the board.

When connected to the board, the environment must be set up by sourcing a specific setup script `fpgas.sh`. This script configures various environment variables based on the targeted FPGA board.

Finally, the bitstream can be uploaded to the FPGA using the `make fpga-program` command, which transfers the top.bit file to the FPGA board. After uploading, deployment of the executable can be achieved with the `make fpga-run` command. By default, this command will use `systest.c` bare-metal application, which prints "Hello from ESP!" on the terminal.

For a specific application the test program must be specified before the command: `TEST_PROGRAM=./soft-build/ariane/baremetal/ita.exe make fpga-run`.

To see the results coming from the FPGA, a tool to print on screen the data coming from the UART serial interface is needed. For this purpose, after logging in `espdev@cs.columbia.edu`, a serial connection can be opened to the FPGA board using the command `minicom -D /dev/vcu118-01 -b 38400`. This allows to observe the results of the test or program running on the FPGA.

3.2 Integrating Multiple ITA Instances: Multi-Head Attention Support

One of the core innovations of Transformer architectures is the ability to run multiple attention heads in parallel, which allows the model to capture diverse aspects of the input data. Since the ITA accelerator processes a single attention head per inference, we explored two potential strategies to support multiple heads, particularly in cases with two or four heads. For both purposes, the application needs some changes.

The first strategy involves reusing a single accelerator instance to handle the attention heads sequentially. This approach prioritizes area efficiency by fitting the heads into the same accelerator but may lead to increased latency due to the sequential processing of heads.

The second strategy involves instantiating separate accelerators, which requires expanding the matrix configuration in the ESP GUI. This method optimizes for reduced latency by allowing concurrent processing of multiple heads, at the cost of increased area usage.

Additionally, in a typical execution of an attention layer, the outputs of multiple heads are concatenated before applying the final linear transformation. However, since the ITA accelerator can only process a single head at a time, this concatenation must be handled in software by the CPU to complete the attention layer's execution. The CPU will sum the outputs of the different heads after they are processed individually by the accelerator. This software-based management of concatenation will introduce additional computational overhead, increasing the overall simulation time due to the extra computational workload.

3.2.1 Multi-heads attention: sequential vs parallel

Now, the two configurations for handling multiple attention heads will be analyzed, starting with the sequential approach, followed by the parallel execution method.

Sequential Execution of Attention Heads

The first solution explored for implementing multi-head attention was the sequential execution of attention heads using a single accelerator instance. This approach was the simplest to implement, as it did not require significant changes to either the existing bare-metal application or the SoC configuration from the ESP GUI. Since the same accelerator was reused for all the attention heads, it allowed us to leverage the existing setup without the need of additional hardware resources or modifications to the system architecture.

In practice, this meant that the heads were processed one after the other, sequentially, leading to a delay, which was alleviated by the better area efficiency. However, the trade-off of this strategy was the increased total processing time due to the sequential execution of the heads, which could impact performance, especially as the number of heads or model complexity increases.

As a first step, using the ITA test generator script `testGenerator.py`, it is necessary to generate the new files required for the computations of the two heads. The input data layout does not change, meaning that the two heads are placed one after the other in the text file `mem.txt`. Consequently, the number of lines in this file simply doubles. The same applies to the golden output file and the other three configuration files for the re-quantization (`rqs_mul.txt`, `rqs_add.txt`, `rqs_shift.txt`).

Following, it's necessary to decide how to handle the data in the bare-metal application. Since the needed data is set in the same way, only varying in size, what changes is the amount of memory needed during allocation. Therefore, it is necessary to modify the constant for the number of heads `N_HEADS` and adjust the memory sizes by multiplying it by the number of heads, as shown in the following code (example for two heads):

```
#define SEQUENCE_LEN 64
#define PROJECTION_SPACE 64
#define EMBEDDING_SIZE 64
#define N_HEADS 2

mem_size = N_HEADS * ( (SEQUENCE_LEN*EMBEDDING_SIZE*2 +
    EMBEDDING_SIZE*PROJECTION_SPACE*4 +
                    PROJECTION_SPACE*3*3 + EMBEDDING_SIZE*3) +
                    (SEQUENCE_LEN*EMBEDDING_SIZE));
gold_size = N_HEADS * SEQUENCE_LEN * EMBEDDING_SIZE;
```

Listing 3.6: Memory allocation adjustment

By increasing the value of `N_HEADS`, memory is allocated for multiple attention heads. This ensures that the system is prepared to handle the increased data requirements of processing multiple heads in sequence.

It is important to ensure that the correct memory offsets are calculated for each head to properly access inputs, weights, and biases. The 16 base pointers used for accessing memory remain the same across the heads. Instead of modifying these pointers, the registers `SRC_OFFSET_REG` and `DST_OFFSET_REG` are utilized to dynamically adjust the input and output memory offsets, allowing the system to read and write data at the correct locations. This approach is advantageous because it avoids the complexity of directly modifying base addresses, allowing the memory structure to remain intact across heads while adjusting the data for each individual head.

The key challenge, then, is to compute the correct memory offsets per head that will be added to the virtual memory base offset. The input and output data for each head must be processed in separate memory regions, which are determined by these offsets. This is where the offset calculations become essential. The following code snippet demonstrates how these offsets are calculated for the multi-head attention mechanism:

```
for(int i = 0; i < N_HEADS; i++) {
    input_offset[i] = i* (SEQUENCE_LEN*EMBEDDING_SIZE*2 +
   EMBEDDING_SIZE*PROJECTION_SPACE*4 + EMBEDDING_SIZE*3*4);
    output_offset[i] = (N_HEADS-1)*(SEQUENCE_LEN*EMBEDDING_SIZE*2 +
   EMBEDDING_SIZE*PROJECTION_SPACE*4 + EMBEDDING_SIZE*3*4) + i * (
   SEQUENCE_LEN*EMBEDDING_SIZE );
}
```

Listing 3.7: Memory offset calculation for multiple heads

In this code, `input_offset[i]` is calculated by multiplying the index of the head, `i`, by the total memory size required for each head's input data. Similarly, `output_offset[i]` calculates the offset where the output data for each head will be written. The term `N_HEADS-1` is used to adjust the starting point of the output memory region based on the total number of head input data, ensuring that each head's output data is placed in the correct location. Specifically, `N_HEADS-1` handles the fact that the memory layout changes when there are multiple heads, and we need to account for this shift in output memory allocation.

As said previously, for this bare-metal application, the same accelerator is used to process each head. However, the accelerator needs to be invoked separately for each head. To do this, a loop is introduced that updates the necessary registers and starts the accelerator for each head in sequence. The following code demonstrates how the accelerator is invoked for each head:

```
for(int heads=0; heads<N_HEADS; heads++) {
    iowrite32(&dev, SRC_OFFSET_REG, input_offset[heads]);
    iowrite32(&dev, DST_OFFSET_REG, output_offset[heads]);

    iowrite32(&dev, ITA_ESP_MULT0_REG, rqs_mul[(heads*2)]);
    iowrite32(&dev, ITA_ESP_MULT1_REG, rqs_mul[(heads*2)+1]);
    iowrite32(&dev, ITA_RIGHT_SHIFT0_REG, rqs_shift[(heads*2)]);
    iowrite32(&dev, ITA_RIGHT_SHIFT1_REG, rqs_shift[(heads*2)+1]);
    iowrite32(&dev, ITA_ADD0_REG, rqs_add[(heads*2)]);
    iowrite32(&dev, ITA_ADD1_REG, rqs_add[(heads*2)+1]);

    iowrite32(&dev, CMD_REG, CMD_MASK_START);
    done = 0;
    while (!done) {
        done = ioread32(&dev, STATUS_REG);
        done &= STATUS_MASK_DONE;
```

```
    }
    iowrite32(&dev, CMD_REG, 0x0);
}
```

Listing 3.8: Accelerator invocation loop per head

As shown in this code, the for loop adjusts the input and output offsets, previously calculated, for each invocation of the accelerator. Additionally, it updates the other registers required for the accelerator, such as ITA_ESP_MULT_REGs, right shifts ITA_RIGHT_SHIFT_REGs, and addition ITA_ADD_REGs. After completing each invocation, the loop waits for the accelerator to signal completion done.

Once this loop is finished, the sum function must be invoked as described earlier. To do this, new pointers need to be generated, which will be passed to the appropriate function, as shown below:

```
u64 *output = (u64 *)aligned_malloc(SEQUENCE_LEN * EMBEDDING_SIZE);
int8_t *mem8_ptr = (int8_t *)mem;
int8_t *output8_ptr = (int8_t *)output;

sum_heads_input(mem8_ptr, output8_ptr);
```

Listing 3.9: Invocation of the Sum Heads Function

In this section of the code, the pointer for output is allocated and then cast to the appropriate types. Then, the function `sum_heads_input` takes the input data (mem8_ptr) and accumulates the results into the output buffer (output8_ptr). It calculates the sum of the outputs from multiple attention heads, as you can see from the code here below:

```
void sum_heads_input(int8_t *mem, int8_t *output) {
    int offset_input = N_HEADS*(SEQUENCE_LEN*EMBEDDING_SIZE*2 +
    EMBEDDING_SIZE*PROJECTION_SPACE*4 + EMBEDDING_SIZE*3*4);
    int8_t sum;
    int output_dimension = SEQUENCE_LEN * EMBEDDING_SIZE;
    for (int i = 0; i < output_dimension; i++) {
        sum = *(mem + offset_input + i) + *(mem + offset_input +
    output_dimension + i);
        *(output+i)= sum;
    }
}
```

Listing 3.10: Sum Heads Function

This function is designed to efficiently sum the outputs from multiple heads by calculating the sum in a single pass through the data. It utilizes a straightforward approach to access and accumulate values from different memory locations.

Parallel Execution of Attention Heads

The second solution explored was the parallel execution of attention heads through the use of multiple instances of the ITA accelerator, with each instance dedicated to a specific head. This approach required modifications to both the ESP SoC configuration and the bare-metal application to enable concurrent execution. With this setup, each attention head could be processed simultaneously by its dedicated accelerator instance, reducing latency.

The first modification involved adjusting the SoC matrix configuration from the ESP GUI to allow several ITA accelerators. To exploit this solution, the number of ITA instances is equivalent to the number of attention heads to be processed. By assigning each head to a separate ITA instance, we ensured that all heads could operate concurrently rather than sequentially. This configuration is illustrated in Fig.3.5, which shows two accelerators, aligned with the number of heads.

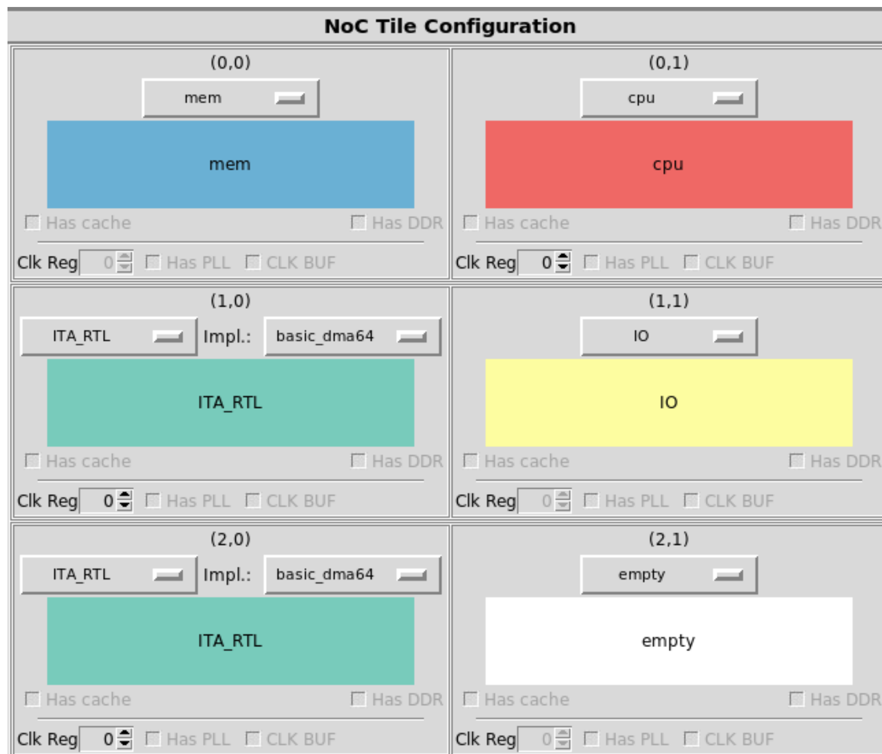

Figure 3.5: ESP GUI with two ITA accelerators

Once the ESP configuration was updated to support this setup, the SoC could effectively manage the concurrent execution of the heads. Now, we'll dive deeper into the modifications made to the bare-metal application.

As in the sequential case, it is essential to adjust the `N_HEADS` parameter

according to the number of heads intended for execution. Furthermore, all pointers to the configuration registers remain unchanged.

On the other hand, in contrast to the sequential processing case, where a single set of input and output files were utilized for all the head data, the current implementation assigns distinct input and output files for each accelerator. Specifically in this case with two accelerators, the files used are `mem_1.txt`, `mem_2.txt`, `gold_1.txt`, and `gold_2.txt`. Each of these files contains the data corresponding to a single head, ensuring that the input and output for each head are processed independently.

In terms of memory allocation, the dimensions for the `malloc` calls are again the same as in the initial case when a single accelerator for a single head was instantiated. This is because, in this scenario, each accelerator will only be invoked once. However, the difference lies in the fact that there are now two separate `malloc` calls, one for each of the two instantiated accelerators. The relevant code snippet for memory allocation is as follows:

```
mem_size = ( (SEQUENCE_LEN*EMBEDDING_SIZE*2 + EMBEDDING_SIZE*
    PROJECTION_SPACE*4 + PROJECTION_SPACE*3*3 + EMBEDDING_SIZE*3) +
    (SEQUENCE_LEN*EMBEDDING_SIZE) );
gold_size = SEQUENCE_LEN * EMBEDDING_SIZE ;

// Memory allocation for input and output
gold_1 = aligned_malloc(gold_size);
gold_2 = aligned_malloc(gold_size);
mem_1 = aligned_malloc(mem_size);
mem_2 = aligned_malloc(mem_size);

// Allocate and populate page table
ptable_1 = aligned_malloc(NCHUNK(mem_size) * sizeof(unsigned *));
ptable_2 = aligned_malloc(NCHUNK(mem_size) * sizeof(unsigned *));
```

Listing 3.11: Memory allocation adjustment

In this code, `mem_size` is calculated based on the required dimensions for input and output data, ensuring that enough memory is allocated for each accelerators. The memory allocation calls to `aligned_malloc` create separate memory blocks for `gold_1`, `gold_2`, `mem_1`, and `mem_2`, ensuring that each accelerator operates independently without interfering with the others.

Additionally, the page table allocations, one for each accelerator, `ptable_1` and `ptable_2` are made using the `NCHUNK` macro to determine the appropriate size for managing memory chunks. Furthermore, the page tables are allocated to facilitate scatter/gather DMA operations, ensuring that data can be accessed efficiently.

The application then includes pre-defined memory data from the header files and probes for available ESP devices. In this case, the probe function will identify two accelerators, given that two instances have been instantiated in the SoC through

the ESP GUI. To configure each detected device, the application employs a for loop that iterates over the number of detected accelerators. Within this loop, several configuration registers are set for each device identified in the *espdevs* array. This process is defined more in detail in the following pseudo-code:

Algorithm 1 Configure Accelerators

1: **for** $n = 0$ **to** $N_HEADS - 1$ **do**
2: $dev \leftarrow \&espdevs[n]$ ▷ Get the device pointer for the current head
3: $coherence \leftarrow ACC_COH_NONE$ ▷ Set coherence based on cache status
4: iowrite32($dev, SELECT_REG$, ioread32($dev, DEVID_REG$))
5: iowrite32($dev, COHERENCE_REG, coherence$)
6: **if** $n == 0$ **then**
7: iowrite32($dev, PT_ADDRESS_REG$, ($unsigned$ long)$ptable_1$)
8: **else**
9: iowrite32($dev, PT_ADDRESS_REG$, ($unsigned$ long)$ptable_2$)
10: **end if**
11: iowrite32($dev, PT_NCHUNK_REG, NCHUNK(mem_size)$)
12: iowrite32($dev, PT_SHIFT_REG, CHUNK_SHIFT$)
13: iowrite32($dev, SRC_OFFSET_REG, 0x0$)
14: iowrite32($dev, DST_OFFSET_REG, 0x0$)
15: iowrite32($dev, ITA_INPUT_PTR_REGS, base_ptr$[0 to 5]) ▷ Input pointers
16: iowrite32($dev, ITA_WEIGHT_PTR_REGS, base_ptr$[2 to 7]) ▷ Weight pointers
17: iowrite32($dev, ITA_BIAS_PTR_REGS, base_ptr$[6 to 9]) ▷ Bias pointers
18: iowrite32($dev, ITA_OUTPUT_PTR_REGS, base_ptr$[10 to 15]) ▷ Output pointers
19: iowrite32($dev, ITA_TILES_REG, tiles$)
20: iowrite32($dev, ITA_ESP_MULT_REGS, rqs_mul[n * 2$ to $n * 2 + 1]$) ▷ Multiplication requests
21: iowrite32($dev, ITA_RIGHT_SHIFT_REGS, rqs_shift[n * 2$ to $n * 2 + 1]$) ▷ Right shift requests
22: iowrite32($dev, ITA_ADD_REGS, rqs_add[n * 2$ to $n * 2 + 1]$) ▷ Addition requests
23: **end for**

In contrast to the sequential processing approach, with multiple ITA accelerators instantiated to handle different heads concurrently, the control flow of the bare-metal application needs to initiate each accelerator individually. This is done by starting each instance, waiting for all to finish, and then resetting their states. The

following code snippet highlights the key modifications needed to manage multiple accelerators in parallel:

```
for(int heads=0; heads<N_HEADS; heads++){
    //Start accelerators
    iowrite32(&espdevs[heads], CMD_REG, CMD_MASK_START);
    //Initialize completion flag
    done[heads] = 0;
}
//Wait for completion of all accelerators
while (!done_all) {
    for(int i=0; i<N_HEADS; i++){
        //Read status and check if done
        done[i] = ioread32(&espdevs[i], STATUS_REG);
        done[i] &= STATUS_MASK_DONE;
        done_all = done_all && done[i];
    }
}
//Reset command registers after completion
for(int i=0; i<N_HEADS; i++){
    iowrite32(&espdevs[i], CMD_REG, 0x0);
}
```

Listing 3.12: Parallel accelerator start and wait loop

Through this code, the accelerators are initiated by looping through the `N_HEADS` devices, issuing the start command (`CMD_MASK_START`) to each of the detected devices. Once all accelerators have been started, the program enters a waiting loop that continuously checks the status registers of each accelerator using the `ioread32` function. The status is masked with `STATUS_MASK_DONE` to determine whether the processing for each head has been completed. The loop exits only when all accelerators have finished their tasks (all heads are done). Then, each command register is reset to `0x0`, ensuring the accelerators are ready for further tasks.

In the end, similarly to the sequential case, the function `sum_heads_input` will be used to finalize the computation of the attention layer by concatenating the outputs from all attention heads. However, in this parallel configuration, the function will require multiple inputs, one for each instantiated accelerator, rather than a single set of inputs as in the sequential case.

Hybrid Approach: Reusing Multiple Accelerators for Multi-Head Execution

In addition to purely sequential or fully parallel approaches, a hybrid method can be employed for implementing multi-head attention. This approach combines the advantages of both methods by reusing a smaller number of accelerators multiple

times, instead of using a single accelerator for sequential processing or dedicating an accelerator to each head for parallel processing.

This hybrid approach reduces the overall hardware resource usage compared to the fully parallel method, while still improving latency over the purely sequential approach. Instead of instantiating all the accelerators or reusing a single accelerator several times, we can instantiate for example half of the accelerators and reuse them to process different heads multiple times.

For instance, with four heads, two accelerators can process one head each in parallel at first, and once this task is completed, the same accelerators are reused to handle the remaining two heads. This allows for a reduction in hardware resources while maintaining a degree of parallel processing that improves timing performance compared to the sequential method.

To implement this hybrid approach in a bare-metal application, several modifications are required. The memory allocations will be the same as in the case of processing two heads sequentially, but this will then be doubled for the second accelerator, as described in the fully parallel strategy.

Additionally, the accelerators must be configured twice, once for each head. After processing the first two heads, the input and output registers are updated to manage the remaining two heads, ensuring that the correct data is loaded and processed during each phase.

This hybrid approach offers several advantages. It reduces the hardware area required by reusing accelerators instead of deploying a dedicated accelerator for each head, which is beneficial in resource-constrained environments. At the same time, it improves latency compared to sequential execution by allowing two heads to be processed simultaneously. This approach also optimizes the use of resources, striking a balance between area efficiency and performance.

3.2.2 Finalizing Layer Acceleration in Transformer Encoder

To effectively accelerate a layer of a Transformer encoder, particularly when excluding the feed-forward layer, two essential operations are required: residual addition and layer normalization.

Residual addition is a crucial component of the Transformer architecture. It allows the output of a sub-layer (self-attention or feed-forward layer) to be added directly to the input of that same sub-layer. This operation helps mitigate the vanishing gradient problem during back-propagation by providing alternative paths for the gradient to flow through the network. As a result, residual connections facilitate deeper network training and improve convergence by preserving the identity function, which allows the model to learn effectively even with multiple layers.

Layer normalization is equally important in this context. It serves to stabilize

and accelerate the training process by normalizing the inputs across the features for each training example. This technique helps maintain a consistent scale and distribution of activations, effectively reducing internal covariate shifts. As a result, layer normalization contributes to achieving faster convergence rates and improves the model's resilience to varying input distributions. Moreover, it facilitates the training of deeper networks by mitigating the effects of non-linearities introduced by activation functions, ensuring that the model learns effectively even as its depth increases.

Since the ITA accelerator is unable to perform these operations directly, it is necessary the development of a software solution that can execute these functions on the processor. Implementing these functions in software can significantly increase execution time compared to the accelerated portions of the model. To address the need for efficient computation, we modified the existing `sum_heads_input` function to incorporate the residual connections and explored libraries or functions that provide optimized implementations for layer normalization.

Specialized libraries tailored for RISC-V architectures, compatible with the Ariane processor, can further enhance the efficiency of mathematical operations. One library is muRISCV-NN[37], which is designed to optimize neural network computations on RISC-V processors developed for embedded platforms and microcontrollers. It is based on ARM's CMSIS-NN library but targets the RISC-V instruction set architecture (ISA) instead. This library provides a set of accelerated kernels that leverage the unique features of the RISC-V architecture, including the RISC-V "V" vector extension v1.0 and the RISC-V packed "P" extension v0.9.6. These extensions enable high-performance vector and packed operations, which are crucial for optimizing neural network computations.

The library offers efficient implementations for various neural network operations, such as matrix multiplication, convolution, and activation functions. By utilizing RISC-V's vector and packed extensions, the library achieves substantial performance improvements. For instance, the matrix multiplication functions are optimized to make full use of RISC-V's vector instructions, resulting in lower latency and higher throughput compared to traditional scalar implementations.

The impact of muRISCV-NN on reducing simulation times is particularly significant in the context of additional computations required for operations like head summation and layer normalization. By offloading these tasks to the optimized library, the CPU's workload is reduced, resulting in improved simulations and quicker performance. For a comprehensive overview of the library and its contributions to optimizing neural network computations on RISC-V, please refer to the paper related[37].

Since we are considering the Ariane RISC-V processor for this SoC, which does not support the vector extension, we can utilize the library to enhance performance, but we will not be able to leverage the full capabilities of vector operations.

Another option is presented in the paper titled `"Optimizing the Deployment of Tiny Transformers on Low-Power MCUs"`[38]. It presents a collection of efficient deep learning kernels specifically designed for resource-constrained environments. This paper highlights strategies and techniques to effectively deploy Transformer models on low-power MCUs.

The GitHub repository associated with this paper offers various pre-implemented functions that have been optimized for performance and memory usage. These implementations not only help in reducing the computational overhead but also ensure that the models can run effectively within the limitations of low-power hardware. By leveraging these resources, we can significantly enhance our implementation efforts, enabling the deployment of Transformer models that are both efficient and scalable. Utilizing these optimized kernels will allow us to address challenges ensuring that our models maintain a high level of performance.

To this end, we decided to take inspiration from the paper's implementation of layer normalization. The implementation of the layer normalization function in the bare-metal application is as follows:

```
void layer_norm(int8_t *input, int8_t *output) {
    float inv_dim = 1/SEQUENCE_LEN;
    int64_t sum = 0;
    int64_t mean = 0;
    int64_t temp = 0;
    int64_t std;
    for(int i = 0; i < SEQUENCE_LEN; i++){
        sum = 0;
        mean = 0;
        for(int j = 0; j < EMBEDDING_SIZE; j++ ){
            mean += *(input + j + i * SEQUENCE_LEN);
        }
        mean = mean / EMBEDDING_SIZE;
        for(int j = 0; j < EMBEDDING_SIZE; j++ ){
            temp = (*(input + j + i * SEQUENCE_LEN)) - mean;
            sum += temp*temp;
        }
        sum = (sum / EMBEDDING_SIZE) + 1;
        plp_sqrt_q64(&sum, 0, &std);
        for(int j = 0; j < EMBEDDING_SIZE; j++){
            *(output + j + i * SEQUENCE_LEN) = (int8_t)( ( *(input + j + i * SEQUENCE_LEN) - mean) /std );
        }
    }
}
```

The function takes two arguments: a pointer to the input tensor of type `int8_t` and a pointer to the output tensor, also of type `int8_t`.

Initially, the function calculates the reciprocal of `SEQUENCE_LEN` and stores it in the variable `inv_dim`. This value will be utilized later for mean calculations. Additionally, several variables are declared and initialized: `sum` and `mean` are both set to zero to accumulate the sum of the elements and to calculate the mean, respectively. The variable `temp` is used to hold intermediate values during the calculations, while `std` will store the standard deviation.

The function proceeds with an outer loop that iterates over each sequence in the input tensor, indexed by `i`. At the start of each iteration, the `sum` and `mean` variables are reset to zero to ensure that calculations for the current sequence do not interfere with previous iterations.

Within the outer loop, the first inner loop computes the mean of the current sequence. It iterates through each embedding dimension, indexed by `j`, and sums the input values corresponding to the current sequence. The mean is then calculated by dividing this cumulative sum by `EMBEDDING_SIZE`.

Following the mean calculation, a second inner loop is employed to compute the sum of the squared differences between each input element and the mean. For each embedding dimension, the temporary variable `temp` holds the difference between the current input element and the mean. This difference is squared and accumulated into the `sum` variable.

After obtaining the sum of squared differences, the function calculates the variance by dividing the accumulated sum by `EMBEDDING_SIZE`. Additionally, 1 is added to the result to prevent potential issues with zero variance. The function then calls `plp_sqrt_q64`, a helper function which will be described later. The result is stored in the variable `std`.

Finally, a third inner loop is executed to normalize each input element. For each embedding dimension, the normalized value is calculated by subtracting the mean from the input element and dividing by the standard deviation. The resulting normalized value is cast to `int8_t` and stored in the corresponding location in the output tensor.

As said before, the `plp_sqrt_q64` function is designed to compute the integer square root of a 64-bit integer using binary search. The implementation is as follows:

```
1  void plp_sqrt_q64(const int64_t *__restrict__ pSrc,
2                   const uint64_t fracBits,
3                   int64_t *__restrict__ pRes) {
4      int64_t number = *pSrc;
5      int64_t root = 0;
6      int64_t start = 0;
7      int64_t end = 46342; // smallest integer that is larger than sqrt
       (0x7FFFFFFF)
8      int64_t mid;
9      if (number > 0) {
```

```
10          while (start <= end) {
11              mid = (start + end) >> 1;
12              if (((mid * mid) >> fracBits) == number) {
13                  root = mid;
14                  break;
15              }
16              if (((mid * mid) >> fracBits) < number) {
17                  start = mid + 1;
18                  root = mid;
19              } else {
20                  end = mid - 1;
21              }
22          }
23          *pRes = root;
24      } else {
25          *pRes = 0;
26      }
27  }
```

This approach allows for improved precision and performance in scenarios where floating-point operations may introduce overhead or inaccuracies.

The function accepts three parameters: a pointer to the input value `pSrc`, a value `fracBits` that specifies the number of fractional bits for fixed-point representation, and a pointer `pRes` where the resulting square root will be stored. It begins by initializing the variable `number` to the value pointed to by `pSrc`, and sets up bounds for a binary search algorithm with `start` initialized to zero and `end` set to 46342, which is the smallest integer greater than the square root of `0x7FFFFFFF`.

If the input number is positive, the function employs a binary search to find its square root. The mid-point is calculated by averaging `start` and `end`, and the algorithm checks if the square of this mid-point matches the input number when adjusted by the specified `fracBits`. If a match is found, the square root is stored in `root`. If the square of the mid-point is less than the input, the search continues in the upper half; otherwise, it narrows down to the lower half.

This method of calculating the square root is superior to a simple floating-point square root calculation because it avoids potential precision loss associated with floating-point arithmetic. Additionally, the binary search approach is computationally efficient, leading to faster execution times in applications that require repeated square root calculations.

3.3 Validation of Accelerator Performance

In order to validate the acceleration capabilities of the designed accelerator, the complete attention layer was implemented using the Ariane processor exclusively. This approach allows for a direct comparison between the performance of the accelerator and the baseline implementation in software, ensuring that any performance improvements can be attributed to the accelerator design.

To obtain this baseline a new application has been developed. It leverages several functions to perform linear transformations, matrix multiplications, and softmax operations essential for implementing a multi-head attention (MHA) layer in a Transformer encoder. The first function, `linear_transform`, is defined as:

```
void linear_transform(int8_t *input, int8_t *output, int8_t *weights,
    int32_t *bias)
```

This function performs a linear transformation by multiplying the input matrix with the weight matrix and adding a bias term. It processes the input and generates an output, ensuring that the results are clamped between -128 and 127, which is suitable for signed 8-bit integer representations. Additionally, `linear_transform_transpose` performs a similar operation but outputs the results in a transposed manner.

The `matmul` function executes matrix multiplication between two matrices, producing the result in a specified output matrix. The function is defined as:

```
void matmul(int8_t *A, int8_t *output, int8_t *C)
```

The final function implemented in the bare-metal application is `iSoftmax`. In this stage, we decided to take inspiration from the paper's implementation [38] of the softmax function. Below is a code snippet that illustrates its implementation:

```
void iSoftmax(
  int8_t * pInBuffer,
  uint8_t * pOutBuffer,
  const int32_t rowDimension,
  const int32_t coeffA,
  const int32_t coeffB,
  const int32_t coeffC,
  const int32_t log2,
  const uint32_t n_levels, int8_t eps_mult_i, int8_t right_shift_i,
    int8_t add_i )
{
    int16_t xTilde;
    int8_t z, p;
    int8_t x_max = -128;
```

```
14      uint32_t y_sum = 0;
15      uint32_t y[rowDimension];
16      // Find the maximum value in the input buffer
17      for (int i=0; i<rowDimension; i++){
18          if (pInBuffer[i] > x_max){
19              x_max = pInBuffer[i];
20          }
21      }
22      // Calculate the exponentials and the sum
23      for (int i=0; i<rowDimension; i++){
24          xTilde = pInBuffer[i] - x_max;
25          z = -(xTilde / log2);
26          p = xTilde + z * log2;
27          y[i] = ((coeffA*(p+coeffB)*(p+coeffB) + coeffC)>>z)*(1-(z >
    31 || z < 0));
28          y_sum += y[i];
29      }
30      // Compute the final softmax output
31      for (int i=0; i<rowDimension; i++){
32          pOutBuffer[i] = (uint8_t)((y[i]*(n_levels-1))/(y_sum));
33      }
34  }
```

It computes the softmax of an input array of integers (`pInBuffer`) and stores the result in an output array (`pOutBuffer`). The function operates on a row of data of size `rowDimension`, adjusting for precision using scaling coefficients (`coeffA`, `coeffB`, `coeffC`) and a logarithmic factor (`log2`). It first calculates the maximum value from the input, then computes the exponentials based on scaled differences. Finally, the function normalizes the values to produce the softmax probabilities across `n_levels` of output, ensuring that each output is appropriately scaled.

For testing this application on FPGA, a 2×2 matrix for the SoC is set from the ESP configuration GUI, which includes a single memory tile, the Ariane processor, an empty tile, and one I/O tile. This setup streamlined the validation process and allowed for a focused assessment of the interactions between the accelerator components effectively.

Chapter 4

Experimental Results

4.1 Baseline Software Performance

The Transformer encoder, widely used in modern deep learning models, is computationally demanding due to its reliance on numerous matrix operations. In particular, the attention mechanism within the encoder performs intensive matrix multiplications, making it a key area of interest when analyzing performance bottlenecks in software implementations. Understanding where time is spent during the execution of the Transformer layer helps guide optimization efforts, particularly with respect to hardware acceleration.

For this reason, a software implementation was implemented. To evaluate its performance, we measured the time spent in each phase of the Transformer encoder running on the Ariane processor. This evaluation focused on the two primary components of the full Transformer layer execution: the attention layer, the residual addition and layer normalization. The results, presented in the following figures and tables, refer to a layer with sizes $S = 64$, $E = 64$, $P = 64$ and one head $H = 1$, and they reveal where computational bottlenecks occur.

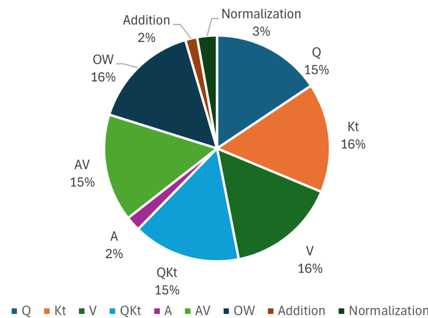

Figure 4.1: Time breakdown for a complete Transformer layer in all its operations

In Fig.4.1, it is evident that a significant portion of time is spent on matrix multiplications within the attention mechanism. While the addition and the normalization steps are together the 5% of the total execution time, the General Matrix Multiplications (GEMMs) dominate the time distribution in this part of the encoder.

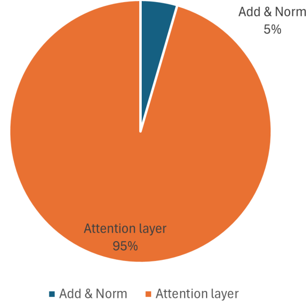

Figure 4.2: Time breakdown for a complete Transformer layer in Attention layer and addition and normalization

The analysis of the total time breakdown split between attention layer, addition and layer normalization, shown in Fig.4.2, further reinforces the dominance of GEMM operations. Following, Tab.4.1 presents the execution time breakdown, detailing the number of clock cycles and corresponding time in milliseconds at a frequency of 78.125 MHz.

Table 4.1: Execution Time Breakdown for a complete Transformer Layer

Operation	Clock Cycles	Time (ms)
Attention Layer	20421938	261.6
Addition & Normalization	968887	12.4
Complete Layer	21390825	274

Approximately 95% of the execution time is consumed by these matrix multiplications. This observation highlights that matrix operations, and so the attention layer, are the primary candidate for acceleration in order to enhance performance.

4.1.1 The Need for Hardware Acceleration

From the results obtained, it becomes clear that the performance of the Transformer encoder in software is largely limited by the time-consuming GEMM operations. Given that these matrix multiplications represent the majority of the computational load, offloading them to specialized hardware is an effective way to improve overall

performance. This leads to the introduction of the ITA accelerator, designed specifically to handle the attention layer computation.

As such, a heterogeneous platform that integrates both the ITA accelerator and the Ariane processor is necessary. In this setup, the accelerator can handle the computationally intensive GEMM operations, while the processor manages auxiliary tasks like addition and normalization. The Embedded Scalable Platform offers the perfect environment for implementing the needed heterogeneity. By enabling the integration of hardware accelerators, ESP provides a flexible and scalable platform to address the specific needs of different stages of the Transformer encoder. With ITA focusing on accelerating GEMM operations and the processor handling the remaining components, the ESP platform ensures a balanced distribution of computational tasks, leading to significant performance gains.

4.2 ITA Acceleration: Performance Evaluation and Analysis

As said before, the integration of specialized accelerators designed to enhance performance through efficient matrix operations is necessary. To address this challenge, we propose the implementation of the ITA accelerator within a 2×2 matrix configuration of ESP system-on-chip. This setup includes the ITA accelerator, the Ariane processor, a memory tile, and an I/O tile, providing a comprehensive environment for optimizing the performance of the attention mechanism while effectively managing data throughput and processing efficiency.

In this context, ITA has been designed to execute a single attention head at a time, excluding the final concatenation of the heads. This design choice allows ITA to focus on optimizing the computationally intensive General Matrix Multiply operations, which are pivotal for Attention calculations. By leveraging ITA, we aim to significantly reduce the execution time of these matrix multiplications, thereby improving the overall efficiency of the Transformer encoder execution time.

The profiling results from the simulation of this setup reveal significant insights into the execution characteristics of the Transformer encoder. Also in this case, the dimensions of the executed layer are $S = 64$, $E = 64$, $P = 64$ and one head $H = 1$. Notably, the majority of the simulation time, approximately 72%, is spent on memory access. This can be attributed to the Direct Memory Access operations, which involve loading and storing data from the main memory to the internal PLMs of the ITA and vice-versa.

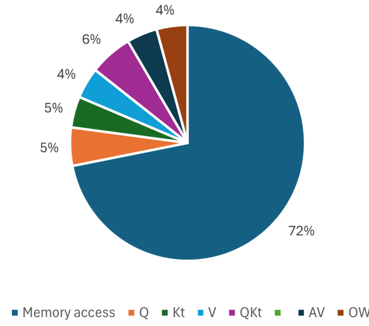

Figure 4.3: ITA acceleration

The results depicted in Fig.4.3 emphasize the bottleneck caused by memory access, indicating that strategies to enhance data transfer efficiency could lead to further performance improvements.

Following the profiling analysis, we provide an overview of the execution of the entire attention layer, where ITA handles the attention computations while the Ariane processor manages the addition and layer normalization. In this configuration, it is observed that approximately 99% of the total execution time is dominated by the operations performed by the Ariane processor. This is primarily due to the overhead associated with managing the normalization and addition, which are executed sequentially after the attention calculations.

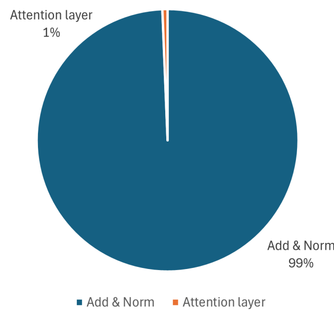

Figure 4.4: Execution breakdown for the total layer with ITA and Ariane processor

The breakdown shown in Fig.4.4 illustrates the execution dynamics, revealing that the attention layer, although partially accelerated by ITA, still incurs significant overhead from the additional operations handled by the main processor. The following table, Tab.4.2, presents the execution time breakdown, detailing the number of clock cycles and corresponding time in milliseconds at a frequency of 78.125 MHz.

Table 4.2: Execution Time Breakdown for a complete Transformer Layer

Operation	Clock Cycles	Time (ms)
Attention Layer	6237	0.08
Addition & Normalization	968887	12.40
Complete Layer	975124	12.48

Comparing the performance of the baseline software implementation to the configuration utilizing the hardware accelerator, we observe a remarkable speed-up of approximately 20x for the execution of a complete Transformer encoder layer. This substantial improvement derives from the highly efficient execution of the attention layer facilitated by ITA, despite the addition and normalization operations remaining in software.

To further illustrate this performance enhancement, we present a graph (Fig. 4.5) comparing the execution times of the software-only baseline with the hardware-accelerated approach using ITA. This comparison visually demonstrates the efficiency gains achieved through the integration of specialized accelerators within the Transformer architecture, highlighting the potential of hardware acceleration to alleviate computational burdens.

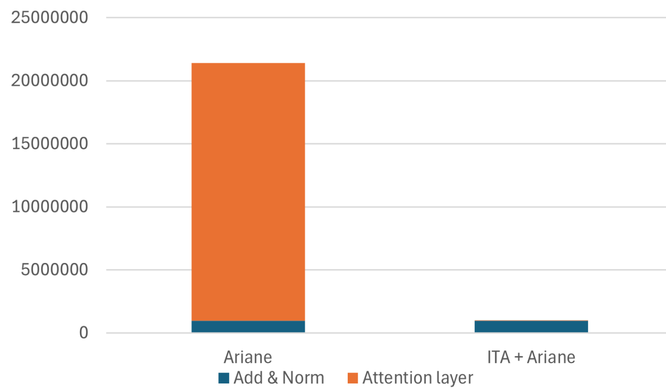

Figure 4.5: Layer acceleration: Ariane vs ITA + Ariane

While the integration of specialized accelerators provides significant speedup in terms of execution time, it is important to note that this comes with an increase in the required hardware resources. Replacing or augmenting a single general-purpose processor with an accelerator inevitably increases the area footprint of the system, as well as its power consumption. This trade-off between execution time and area/power is crucial when considering hardware acceleration solutions in a limited resources environment.

In the case of the ESP system-on-chip deployed on an FPGA, the area overhead introduced by the ITA, in comparison to a standard processor like Ariane, must be carefully evaluated. The ITA accelerator is designed specifically to handle matrix multiplications and attention layer operations with high parallelism, which means it consumes significantly more resources than a general-purpose processor performing these tasks.

Through synthesis and implementation using Vivado 2023.2, we can quantify the area requirements before generating the final bitstream for the FPGA. This step provides detailed information on the contributions of both the Ariane processor and, more notably, the ITA accelerator in terms of logic cells, DSP blocks, and other critical FPGA resources.

Figure 4.6: Area breakdown showing the contributions of Ariane processor and ITA accelerator in FPGA deployment.

As illustrated in Fig.4.6, the accelerator, which is the component in orange, consumes a significant portion of the available FPGA resources due to its specialized architecture. This contrasts with the more modest area footprint of the Ariane processor, which is the element in purple. The trade-off becomes evident: the speed-up achieved by ITA in terms of timing performance is directly related to the increase in area resources.

Thus, if the objective is to achieve substantial speed-up in executing attention layers, one must account for the additional area required to accommodate a specialized accelerator. In practical terms, this means balancing the desired performance improvements with the resource constraints imposed by the FPGA platform.

Moreover, as this analysis shows, the area trade-offs for deploying an SoC on FPGA are highly dependent on the complexity of the tasks being offloaded to accelerators. In this specific deployment scenario, Vivado's synthesis and implementation reports provide a comprehensive overview of how much additional area ITA requires relative to the Ariane processor.

In an FPGA-based SoC implementation, it is crucial to quantify the hardware

resources required by each component, especially when integrating specialized accelerators like ITA. The area of an FPGA is typically expressed in terms of several key metrics: lookup tables (LUTs), flip-flops (often referred to as registers or REGs), block RAM (BRAM), and digital signal processing units (DSPs). Each of these elements plays a crucial role in determining the overall hardware footprint of the deployed system. LUTs are used for implementing logic functions, REGs serve as storage elements, BRAM provides on-chip memory for storing intermediate data, and DSPs are specialized units optimized for arithmetic operations like multiplications and additions.

When comparing the ITA accelerator with the Ariane processor on FPGA, the data extracted from the Fig. 4.7) shows a significant difference in the resource footprint between the two components.

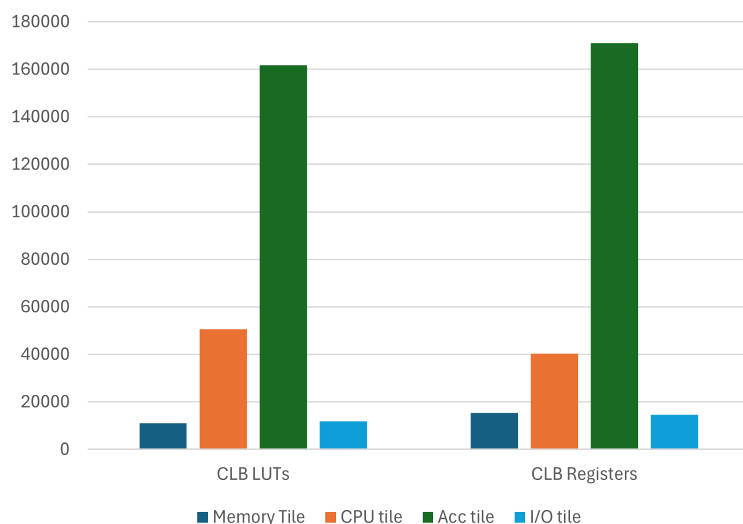

Figure 4.7: FPGA resource allocation

The Fig.4.7 presents a comprehensive overview of all four distinct tiles present in our SoC: the Ariane processor, the ITA accelerator, the memory tile, and the I/O tile. In detail, it highlights that the ITA accelerator consumes substantially more resources than the Ariane processor, particularly in terms of LUTs. This is expected given the ITA's primary role in accelerating General Matrix Multiply operations, which are computationally intensive and require significant hardware support for efficient execution.

In contrast, the Ariane processor, which is a general-purpose RISC-V core, consumes fewer resources since its tasks in the SoC are primarily control-oriented rather than computationally heavy. This difference becomes even more pronounced when considering the CLB (Configurable Logic Block) usage, which was the primary

metric analyzed due to the stark contrasts observed between the two components. The CLBs represent the FPGA's basic building blocks, and the ITA accelerator requires far more of these to perform its specialized operations.

Furthermore, the analysis reveals that the Ariane processor uses 27 DSPs, while the ITA accelerator uses only 22. This outcome is unexpected, as ITA, being specifically designed for matrix multiplication tasks, would be expected to utilize more DSP resources to better support the intensive computation required.

In addition to the overall resource comparison, we delve deeper into the ITA accelerator to analyze the resource consumption of its internal components, as shown in Fig.4.8.

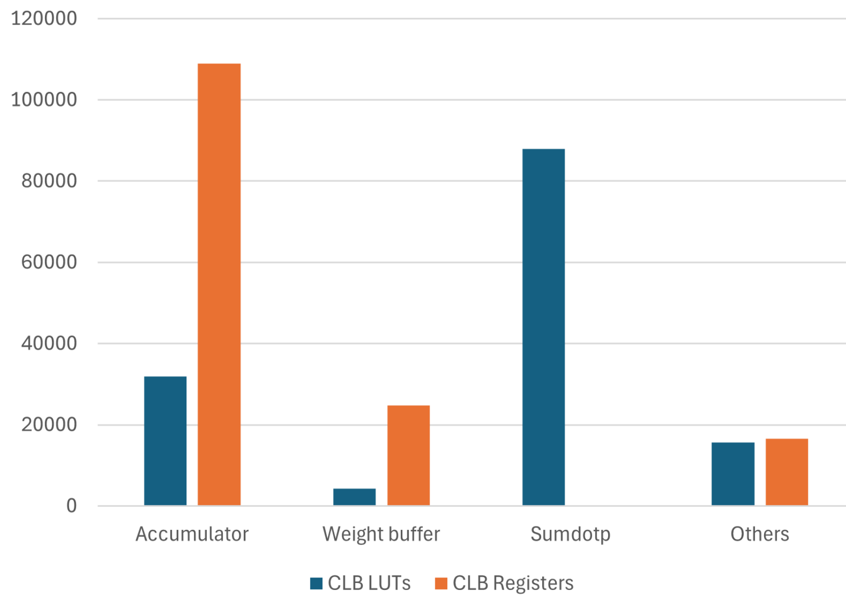

Figure 4.8: Resource breakdown of the ITA accelerator on FPGA

The Fig. 4.8 shows that the ITA's internal components consume. It reveals significant insights into the resource utilization of its various components. Among these, the accumulator stands out as the most resource-intensive element, consuming a considerable portion of the register resources.

In contrast, the sumdotp component is notable for its high utilization of Look-Up Tables (LUTs). This component is essential for performing the dot product calculations required during the execution of matrix multiplications.

When considering the remaining components of the accelerator, collectively labeled as "others," it becomes clear that the accumulator, the weight buffer, and the sumdotp are the most complex parts of the ITA architecture, demanding the highest resources. This concentration of resource utilization in these key components

is indicative of their fundamental roles in ensuring the efficient operation of the ITA accelerator within the FPGA environment, reinforcing the necessity for careful resource management when deploying specialized accelerators in system-on-chip designs.

4.3 Optimizing the Attention Mechanism with Multiple Accelerators

The attention mechanism in Transformers, as said before, is computationally intensive, particularly in large models with multiple attention heads. One potential optimization is to parallelize the computation of these attention heads using multiple accelerators. This is made feasible by leveraging the ESP platform, which allows for easy instantiation of additional accelerators.

In ESP, adding more accelerators involves simply increasing the size of the matrix configuration of the SoC from the ESP GUI. The platform's flexible architecture enables the seamless integration of multiple ITA accelerators, which can process different heads in parallel.

4.3.1 Parallelization Strategies: Sequential vs. Parallel Execution

In a sequential configuration, a single ITA accelerator handles all attention heads one after another. In contrast, using the power of the ESP platform, parallel execution involves assigning multiple heads to different ITA accelerators, which compute them simultaneously. Although parallelization intuitively suggests that the total execution time should decrease significantly, this is not always the case in practice. One of the primary bottlenecks is memory access latency, especially when multiple accelerators compete for access to shared resources, like memory, via DMA.

We present the performance data for both strategies using simulations for two heads and four heads at a frequency of 78.125 MHz.

4.3.2 Two Heads: Sequential vs. Parallel Execution

In the case of two heads, we simulated both sequential and parallel execution. Tab.4.3 shows the profiling data for sequential execution with a single ITA accelerator, while Tab.4.4 presents the results for parallel execution using two ITA accelerators.

Table 4.3: Two Heads - Sequential Execution (1 ITA accelerator)

Section	Execution Time (ns)	Clock Cycles
Memory Access (Head 1)	57420.8	4486
Computation (Head 1)	22617.6	1767
Memory Access (Head 2)	57382.4	4483
Computation (Head 2)	22617.6	1767
Total	171481.6	13397

Table 4.4: Two Heads - Parallel Execution (2 ITA accelerators)

Section	Execution Time (ns)	Clock Cycles
Memory Access (Head 1)	100492.8	7851
Computation (Head 1)	22617.6	1767
Memory Access (Head 2)	93004.8	7266
Computation (Head 2)	22617.6	1767
Total	129369.6	9618

As seen in Tab.4.3, the total execution time for sequential execution is 171481.6 ns. In contrast, the parallel execution time is reduced to 129369.6 ns (Tab.4.4), corresponding to a speedup of 1.32×. It is important to note that the parallel execution time is not halved as expected. This is due to increased memory contention as both accelerators request data from memory simultaneously, causing few delays.

4.3.3 Four Heads: Sequential vs. Parallel Execution

The performance difference becomes even more pronounced when scaling to four heads. In the sequential case, a single ITA accelerator processes each head one by one, as shown in Tab.4.5. Each head exhibits consistent execution times, taking the same amount of time to process. Therefore, the total execution time for four heads will be four times the execution time of a single head. However, this total time is increased slightly due to the overhead incurred when re-configuring the accelerator via the bare-metal application and adjusting the configuration registers, which vary from one head to another. The overall execution time when processing all four heads sequentially amounts to 346393.6 ns with a total of 27500 clock cycles.

Table 4.5: Four Heads - Sequential Execution (1 ITA accelerator)

Section	**Execution Time (ns)**	**Clock Cycles**
Memory Access	57382.4	4483
Computation	22617.6	1767
Total (1 Head)	80000.0	6250
Total (4 Heads)	346393.6	27500

To improve performance, two ITA accelerators can be utilized to handle the processing of the heads in parallel. When parallelizing with two ITA accelerators, the execution time is reduced to 280832 ns (Tab.4.6), corresponding to a speedup of 1.23×. Similar to the two-head case, the reduction is less than expected due to memory access contention.

Table 4.6: Four Heads - Parallel Execution (2 ITA accelerators)

Section	**Exec. Time (ns)**	**Clock Cyc.**
Head 1 - Memory Access (1st Acc.)	100364.8	7841
Head 1 - Computation (1st Acc.)	22617.6	1767
Head 2 - Memory Access (1st Acc.)	100364.8	7841
Head 2 - Computation (1st Acc.)	22617.6	1767
Total (1st Acc.)	122982.4	9608
Head 1 - Memory Access (2nd Acc.)	99750.4	7793
Head 1 - Computation (2nd Acc.)	22617.6	1767
Head 2 - Memory Access (2nd Acc.)	99622.4	7783
Head 2 - Computation (2nd Acc.)	22617.6	1767
Total (2nd Acc.)	122368	9560
Total (Both Acc.)	280832	21403

Lastly, when employing a fully parallel configuration with four instances of the ITA accelerator, each head is processed simultaneously. This results in a total execution time of 219865.6 ns for all four heads combined, significantly enhancing performance compared to both sequential and dual-accelerator configurations. In this case it corresponds to a speedup of 1.58×.

Table 4.7: Four Heads - Fully Parallel Execution (4 ITA accelerators)

Section	Execution Time (ns)	Clock Cycles
Head 1 - Memory Access	182732.8	14276
Head 1 - Computation	22617.6	1767
Total for Head 1	205350.4	16043
Head 2 - Memory Access	177241.6	13847
Head 2 - Computation	22617.6	1767
Total for Head 2	199859.2	15614
Head 3 - Memory Access	176665.6	13803
Head 3 - Computation	22617.6	1767
Total for Head 3	199283.2	15569
Head 4 - Memory Access	163635.2	12784
Head 4 - Computation	22617.6	1767
Total for Head 4	186252.8	14551
Overall Total	219865.6	17502

So, parallelizing the execution of attention heads using multiple accelerators on ESP offers clear performance benefits, but the improvement is limited by memory access bottlenecks. The results highlight that while parallel execution can reduce computation time, the contention for shared memory resources introduces delays, particularly as the number of accelerators increases.

4.3.4 Reduced Benefits as Parallelization Increases

As more ITA accelerators are added, a significant reduction in computation time is observed initially. However, as demonstrated in the previous sections, this reduction does not scale linearly with the number of accelerators. This is because parallelizing beyond a certain point leads to saturation, where the performance gains become marginal due to increasing memory access contention.

For example, in the case of two heads, the parallel execution time was reduced by approximately 24.5%. However, when scaling to four heads, the improvement dropped to around 19% with two accelerators and 36.5% with four accelerators. The diminishing returns occur because each additional accelerator introduces more competition for shared memory, which becomes the dominant bottleneck rather than computation.

Moreover, these speed-ups only refer to the accelerated portion of the layer. Since, as shown in Fig. 4.5, once the attention computation is accelerated, the total time is dominated by addition and normalization (still executed in software). Consequently, the speedup obtained by introducing more instances of ITA, being

very little, will not affect the final speed-up of the entire layer, which will remain almost the same as before.

Nonetheless, these analyses still offer interesting insights, because nothing prevents, in principle, from also accelerating addition and normalization (e.g. using a separate accelerator tile). In that scenario, evaluating the maximum speedup achievable by parallelizing multiple attention heads on ITA, and the saturation effect due to memory conflicts, becomes relevant.

4.3.5 Pareto Optimal Trade-offs Between Latency and Area

In hardware design, achieving Pareto optimality involves balancing trade-offs between multiple factors, such as execution time (latency) and the resources (area) consumed by the system. Adding more accelerators certainly improves execution time, but it also increases the area footprint and power consumption of the system. At some point, further increases in the number of accelerators will not provide a proportional decrease in latency, and the design will no longer be Pareto efficient.

In our case, as more accelerators are added to handle the attention heads, the latency initially decreases, but the area and resource usage increase. The relationship between these two factors can be illustrated using a Pareto curve. For the 4-head simulation, the data suggests that beyond two accelerators, the system approaches a point of diminishing returns. This is evident when considering that the execution time reduction from two to four heads was much smaller than from one to two heads, while resource consumption continued to grow linearly.

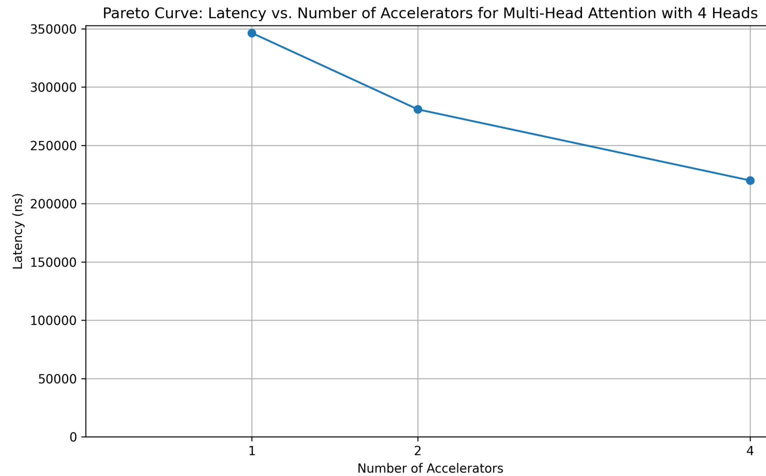

Figure 4.9: Pareto Curve: Latency vs. Area for Multi-Head Attention with 4 Heads

The Pareto curve in Fig.4.9 illustrates the trade-off between latency and area for different numbers of attention heads and ITA accelerators.

From the data, it is clear that increasing the number of ITA accelerators provides diminishing performance gains beyond a certain point due to memory contention. While parallelization can yield significant improvements in execution time, this approach reaches a saturation point where adding more accelerators no longer justifies the additional area overhead.

The Pareto curve demonstrates that an optimal balance between latency and area must be considered when designing systems with multiple accelerators. In our simulations, the sweet spot appears to be around two to four accelerators, where the trade-off between performance improvement and area consumption is most favorable. Further scaling of the system would require architectural changes, such as improved memory hierarchies or more efficient resource management, to overcome the bottlenecks imposed by memory contention.

Chapter 5

Conclusions and future works

This thesis presented the integration and evaluation of the ITA accelerator within the ESP system, targeting the acceleration of the Encoder part in Transformer models. Through simulation, ITA demonstrated a significant 20x speed-up compared to the software-only baseline running on the Ariane processor. This performance boost was achieved by offloading the General Matrix Multiplication (GEMM) operations in the attention layer to the accelerator while leaving the addition and layer normalization operations to the processor.

The deployment of ITA within ESP shows the clear advantage of leveraging hardware accelerators for computationally demanding operations like the attention mechanism. Additionally, the flexibility of ESP allows for easy scalability, potentially enabling more advanced optimizations such as parallelizing multiple attention heads using additional accelerator instances. Despite this success, the evaluation was limited to simulations due to challenges encountered during FPGA deployment, which highlights areas for future work.

In fact, one of the key challenges faced during this project was the inability to fully deploy the accelerator on the FPGA. The bitstream generation encountered timing issues, requiring the processor frequency had to be reduced from 78.125 MHz to 25 MHz to meet timing constraints. Although the bitstream was eventually generated, the accelerator failed to complete its execution when tested on the FPGA. This issue is likely due to the design being optimized for ASIC rather than FPGA, leading to a mismatch in resource requirements. Future work should focus on resolving these issues, ensuring that the accelerator is properly synthesized for FPGA deployment, and achieving successful execution on hardware.

Another area of exploration involves the parallelization of the attention heads using multiple instances of ITA within ESP. While the attention layer achieved

significant speedup, the overall execution time is still dominated by the addition and normalization operations handled by the Ariane processor. Theoretical scaling could be achieved by instantiating additional accelerators specialized to offload these bottleneck functions. This approach could leverage ESP's scalable architecture, which supports the instantiation of additional processing, memory, or accelerator tiles as needed.

A possible scaling model could focus on distributing the computation of these functions across multiple cores. The function responsible for summing attention heads could be parallelized by splitting the input across multiple processors, allowing each core to compute part of the summation. Similarly, the layer_norm function could be distributed by assigning different sequence lengths to each core, thus speeding up the mean and variance calculations.

For example, if the number of heads or the sequence length grows, additional processors could handle these portions of the workload. Ideally, one processor per head or chunk of sequences would ensure linear scaling of the operations, theoretically reducing the execution time by a factor proportional to the number of processors instantiated.

However, adding more processors introduces challenges, such as increased memory traffic and congestion, especially as multiple cores compete for shared memory access. This could be mitigated by introducing additional memory tiles within ESP, providing more bandwidth to handle the concurrent memory accesses without significant degradation in performance.

In addition to this theoretical scaling with processors, another promising avenue for future optimization is maximizing ITA's workload capacity through the use of tiling techniques. It would be possible to better exploit its processing power, thereby improving throughput. This strategy would involve adjusting the size of the tiles to fit within the internal PLM (Private Local Memory) of ITA while ensuring that larger data sets can be processed efficiently.

However, the first challenge in this approach is to modify the HWPE interface used to integrate the ITA. To fully leverage tiling, the HWPE interface would need to support a different data streaming, allowing the accelerator to process data continuously as it is fed in chunks. Such a modification would reduce idle time and maximize the use of the accelerator's computational resources.

Another consideration in this approach is the internal PLM of ITA, which would need to be carefully managed to hold a larger quantity of data. As the tile size increases, the PLM must accommodate more data while still ensuring that it does not become a bottleneck. The size of the PLM would directly influence the performance gains achievable through tiling. Thus future work should explore how best to balance tile size and memory capacity to maximize efficiency.

Future work could formalize a theoretical model to better understand the scaling impact of these additions. The model would need to account for the performance

trade-offs between increased parallelization and potential resource contention, particularly with respect to memory access. Simulating the effect of different numbers of processing cores and memory tiles could provide valuable insights into achieving an optimal balance between resource allocation and performance enhancement.

In conclusion, while this work demonstrated the potential of ITA for accelerating Transformer layers, significant opportunities remain for optimization and deployment, particularly on FPGA hardware. Resolving the FPGA issues and exploring parallelization strategies could lead to even greater performance improvements, further enhancing the capabilities of hardware-accelerated deep learning models.

Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. «Attention Is All You Need». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Curran Associates, Inc., Dec. 2017, pp. 5998–6008. DOI: 10.48550/arXiv.1706.03762. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf (cit. on pp. 1, 6, 7, 10).

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». In: *arXiv preprint arXiv:1810.04805* (2018). URL: https://arxiv.org/abs/1810.04805 (cit. on p. 1).

[3] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. «Improving Language Understanding by Generative Pre-Training». In: *OpenAI Blog* (2018). URL: https://www.openai.com/research/language-unsupervised (cit. on p. 1).

[4] Alexey Dosovitskiy et al. «An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale». In: *arXiv preprint arXiv:2010.11929* (2020). URL: https://arxiv.org/abs/2010.11929 (cit. on pp. 1, 9).

[5] Gamze Islamoglu, Moritz Scherer, Gianna Paulin, Tim Fischer, Victor J.B. Jung, Angelo Garofalo, and Luca Benini. «ITA: An Energy-Efficient Attention and Softmax Accelerator for Quantized Transformers». In: *2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. Vol. 65. IEEE, Aug. 2023, pp. 1–6. DOI: 10.1109/islped58423.2023.10244348. URL: http://dx.doi.org/10.1109/ISLPED58423.2023.10244348 (cit. on pp. 2, 11, 16, 17).

[6] *ESP - The Open Source SoC Platform*. Website. Accessed: 2024-09-13. URL: https://www.esp.cs.columbia.edu (cit. on pp. 2, 16).

[7] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. «Agile SoC Development with Open ESP». In: *2020 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9 (cit. on pp. 2, 16, 20).

[8] *CS231n Convolutional Neural Networks for Visual Recognition*. `https://cs231n.github.io/neural-networks-1/` (cit. on p. 5).

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016 (cit. on p. 5).

[10] Susmita Das, Amara Tariq, Thiago Santos, Sai Sandeep Kantareddy, and Imon Banerjee. «Recurrent Neural Networks (RNNs): Architectures, Training Tricks, and Introduction to Influential Research». In: *Machine Learning for Brain Disorders*. Ed. by Olivier Colliot. New York, NY: Springer US, 2023, pp. 117–138. ISBN: 978-1-0716-3195-9. DOI: `10.1007/978-1-0716-3195-9_4`. URL: `https://doi.org/10.1007/978-1-0716-3195-9_4` (cit. on p. 6).

[11] Sepp Hochreiter. «The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions». In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 06.02 (1998), pp. 107–116. DOI: `10.1142/S0218488598000094`. URL: `https://doi.org/10.1142/S0218488598000094` (cit. on p. 6).

[12] Robin M. Schmidt. *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview*. 2019. arXiv: `1912.05911 [cs.LG]`. URL: `https://arxiv.org/abs/1912.05911` (cit. on p. 6).

[13] Ralf C. Staudemeyer and Eric Rothstein Morris. *Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks*. 2019. arXiv: `1909.09586 [cs.NE]`. URL: `https://arxiv.org/abs/1909.09586` (cit. on p. 6).

[14] Shaked Brody, Uri Alon, and Eran Yahav. «On the Expressivity Role of LayerNorm in Transformers' Attention». In: *Findings of the Association for Computational Linguistics: ACL 2023*. 2023, pp. 14211–14221. DOI: `10.18653/v1/2023.findings-acl.895`. URL: `https://aclanthology.org/2023.findings-acl.895` (cit. on p. 8).

[15] Dan Hendrycks and Kevin Gimpel. «Gaussian Error Linear Units (GELUs)». In: *arXiv preprint arXiv:1606.08415* (2016). Trimmed version of 2016 draft. URL: `https://doi.org/10.48550/arXiv.1606.08415` (cit. on p. 12).

[16] Hang Xiao, Haobo Xu, Xiaoming Chen, Yujie Wang, and Yinhe Han. «Fast and High-Accuracy Approximate MAC Unit Design for CNN Computing». In: *IEEE Embedded Systems Letters* 14.3 (2022), pp. 155–158. DOI: `10.1109/LES.2021.3137335` (cit. on p. 13).

[17] Pudi Dhilleswararao, Srinivas Boppu, M. Sabarimalai Manikandan, and Linga Reddy Cenkeramaddi. «Efficient Hardware Architectures for Accelerating Deep Neural Networks: Survey». In: *IEEE Access* 11 (2023), pp. 76580–76595. DOI: 10.1109/ACCESS.2023.3298835. URL: https://www.researchgate.net/publication/366316225_Efficient_Hardware_Architectures_for_Accelerating_Deep_Neural_Networks_Survey (cit. on pp. 13–15).

[18] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh. «Neural Acceleration for GPU Throughput Processors». In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 482–493 (cit. on p. 14).

[19] Janardan Misra and Indranil Saha. «Artificial neural networks in hardware: A survey of two decades of progress». In: *Neurocomputing* 74.1 (2010). Artificial Brains, pp. 239–255. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2010.03.021. URL: https://www.sciencedirect.com/science/article/pii/S092523121000216X (cit. on p. 14).

[20] Wim Vanderbauwhede and Khaled Benkrid. *High-performance computing using FPGAs*. Mar. 2014, pp. 1–803. ISBN: 978-1-4614-1790-3. DOI: 10.1007/978-1-4614-1791-0 (cit. on p. 14).

[21] Norman P. Jouppi, Cliff Young, Nikhil Patil, et al. «In-Datacenter Performance Analysis of a Tensor Processing Unit». In: *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 1–12. DOI: 10.1109/ISCA.2017.8000215. URL: https://arxiv.org/abs/1702.04773 (cit. on pp. 14, 15).

[22] Cristina Silvano et al. *A Survey on Deep Learning Hardware Accelerators for Heterogeneous HPC Platforms*. 2024. arXiv: 2306.15552 [cs.AR]. URL: https://arxiv.org/abs/2306.15552 (cit. on p. 15).

[23] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. «[DL] A Survey of FPGA-based Neural Network Inference Accelerators». In: *ACM Transactions on Reconfigurable Technology and Systems* 12 (Mar. 2019), pp. 1–26. DOI: 10.1145/3289185 (cit. on p. 15).

[24] R. Machupalli, M. Hossain, and M. Mandal. «Review of ASIC accelerators for deep neural network». In: *Microprocessors and Microsystems* 89 (2022), p. 104441 (cit. on p. 15).

[25] Sparsh Mittal and Jeffrey S. Vetter. «A survey of CPU-GPU heterogeneous computing techniques». In: *ACM Computing Surveys (CSUR)* 47.4 (2015), p. 69 (cit. on p. 15).

[26] Junki Park, Hyunsung Yoon, Daehyun Ahn, Jungwook Choi, and Jae-Joon Kim. «OPTIMUS: OPTImized matrix MUltiplication Structure for Transformer neural network accelerator». In: *Conference on Machine Learning and Systems*. 2020. URL: https://api.semanticscholar.org/CorpusID:219850852 (cit. on p. 15).

[27] Hanrui Wang, Zhekai Zhang, and Song Han. «SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning». In: *CoRR* abs/2012.09852 (2020). arXiv: 2012.09852. URL: https://arxiv.org/abs/2012.09852 (cit. on p. 15).

[28] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W. Lee. «ELSA: Hardware-Software Co-design for Efficient, Lightweight Self-Attention Mechanism in Neural Networks». In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 692–705. DOI: 10.1109/ISCA52012.2021.00060 (cit. on p. 15).

[29] *HWPE Documentation*. https://hwpe-doc.readthedocs.io/en/latest/modules.html (cit. on pp. 18, 19).

[30] Luca Benini, Davide Rossi, et al. *PULP Platform: Parallel Ultra Low Power*. https://github.com/pulp-platform. Available at: GitHub and ETH Zurich and University of Bologna. 2024. URL: https://www.pulp-platform.org/ (cit. on p. 18).

[31] Philip Wiese, Gamze İslamoğlu, Moritz Scherer, Luka Macan, Victor J. B. Jung, Alessio Burrello, Francesco Conti, and Luca Benini. *Toward Attention-based TinyML: A Heterogeneous Accelerated Architecture and Automated Deployment Flow*. 2024. arXiv: 2408.02473 [cs.AR]. URL: https://arxiv.org/abs/2408.02473 (cit. on p. 18).

[32] *Ariane*. https://github.com/openhwgroup/cva6 (cit. on p. 21).

[33] D. Giri, K.-L. Chiu, G. Eichler, P. Mantovani, and L.P. Carloni. «Accelerator Integration for Open-Source SoC Design». In: *IEEE Micro* 41.4 (2021), pp. 33–40. DOI: 10.1109/MM.2021.3077543. URL: https://ieeexplore.ieee.org/document/9406354 (cit. on p. 21).

[34] *How to: design an accelerator in RTL*. Tech. rep. Accessed: 2024-09-13. Columbia University, 2024. URL: https://www.esp.cs.columbia.edu/docs/rtl_acc/rtl_acc-guide/ (cit. on p. 22).

[35] ESP Team. *ESP Accelerator Specification*. Accessed: 2024-09-13. Columbia University. Mar. 2022. URL: https://www.esp.cs.columbia.edu/docs/specs/esp_accelerator_specification.pdf (cit. on pp. 23, 24, 26, 27).

[36] Luca P. Carloni. «From Latency-Insensitive Design to Communication-Based System-Level Design». In: *Proceedings of the IEEE* 103 (2015), pp. 2133–2151. URL: https://api.semanticscholar.org/CorpusID:10898238 (cit. on p. 25).

[37] Philipp van Kempen, Jefferson Parker Jones, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. «muRISCV-NN: Challenging Zve32x Autovectorization with TinyML Inference Library for RISC-V Vector Extension». In: *Proceedings of the 21st ACM International Conference on Computing Frontiers Workshops and Special Sessions*. CF '24 Companion. Ischia, Italy: Association for Computing Machinery, 2024, pp. 75–78. DOI: 10.1145/3637543.3652878. URL: https://doi.org/10.1145/3637543.3652878 (cit. on p. 55).

[38] Victor J. B. Jung, Alessio Burrello, Moritz Scherer, Francesco Conti, and Luca Benini. *Optimizing the Deployment of Tiny Transformers on Low-Power MCUs*. 2024. arXiv: 2404.02945 [cs.LG]. URL: https://arxiv.org/abs/2404.02945 (cit. on pp. 56, 59).