Hardware Accelerator for Transformer based End-to-End Automatic Speech Recognition System

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
in
Electronics and Communication Engineering by Research

by

D Shaarada Yamini
2020702003
yamini.d@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, INDIA
September 2023

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled “Hardware Accelerator for Transformer based End-to-End Automatic Speech Recognition System” by D Shaarada Yamini, has been carried out under my supervision and is not submitted elsewhere for a degree.

_____ _____

Date Adviser: Prof. Suresh Purini

To my family and friends for always believing in me.

Acknowledgments

I am grateful to my advisor, Dr. Suresh Purini, for his guidance and unwavering support throughout the project. Along with being an invaluable mentor and a resource for technical and research expertise, he was also extremely kind and understanding throughout my research journey, for which I will remain indebted. I thank my co-advisor, Dr. Anil Kumar Vuppala, for his supportive role in the speech lab and for his insights and guidance along the way. A heartfelt thanks to my Ph.D. mentor, Ganesh Mirishkar, for his significant contributions to the speech side of the project and the software implementations of the model throughout the entire duration. I am grateful for your patience and willingness to address my inquiries whenever needed.

I would like to express my deepest gratitude to my parents, my beloved sister Sivani, and my supportive husband Girish for their unwavering support, patience, and love throughout my academic journey. Their belief in me has been invaluable in completing this thesis.

I would also like to extend my heartfelt appreciation to my dearest friends, Dheepika and Teja, to name a few, who were a source of encouragement and motivation, and for making this journey memorable.

Abstract

Hardware accelerators are being designed to offload compute-intensive tasks such as deep neural networks from the CPU to improve the overall performance of an application, specifically on the performance-per-watt metric. With the evolution of speech recognition based applications, many deep learning models for Automatic Speech Recognition have been proposed. Encoder-decoder-based sequence-to-sequence models such as the Transformer model have demonstrated state-of-the-art results in end-to-end automatic speech recognition systems (ASRs). However, the Transformer model being intensive on memory and computation poses a challenge for an FPGA implementation.

This thesis proposes an end-to-end architecture to accelerate a Transformer for an ASR system. The host CPU orchestrates the computations from different encoder and decoder stages of the Transformer architecture on the designed hardware accelerator with no necessity for intervening FPGA reconfiguration. The intermediate stages, like data pre-processing and feature extraction, are performed on the host while the complex recognizer, i.e., the Transformer model, is offloaded onto an FPGA. The communication latency is hidden by prefetching the weights of the next encoder/decoder block while the current block is being processed. The larger computations in the model are split across both the Super Logic Regions (SLRs) of the FPGA, mitigating the inter-SLR communication. The proposed design presents an optimal latency, exploiting the available resources. The accelerator design is realized using Vitis high-level synthesis tool, using OpenCL, a language for heterogeneous computing, and evaluated on an Alveo U-50 FPGA card. The end-to-end ASR system has a latency of $\sim$120ms, which is suitable for real-time applications. The design demonstrates an average speed-up of $32\times$ compared to an Intel Xeon E5-2640 CPU and $8.8\times$ compared to NVIDIA GeForce RTX 3080 Ti Graphics card for a 32-bit floating point single precision model.

Contents

List of Figures

List of Tables

List of Abbreviations

ASIC Application Specific Integrated Circuits

ASR Automatic Speech Recognition

AXI Advanced eXtensible Interface

BRAM Block Random Access Memory

CLB Configurable Logic Block

CNN Convolutional Neural Network

CPU Central Processing Unit

DNN Deep Neural Network

DSP Digital Signal Processor

FFN Feed Forward Network

FPGA Field Programmable Gate Array

GFLOPs Giga Floating Point Operations

GPU Graphics Processing Unit

HBM High Bandwidth Memory

HMM Hidden Markov Model

IoT Internet of Things

LUT Look Up Tables

MAC Multiply and Accumulate

MHA Multi Headed Attention

PAL Programmable Array Logic

PCIe Peripheral Component Interconnect Express

RNN Recurrent Neural Network

Chapter 1

Introduction

1.1 Motivation

The recent trends in personal digital assistants and smart home technologies have led to extensive use of speech recognition technology. Automatic speech recognition (ASR) involves transcribing a speech signal into text which is perhaps consumed by downstream modules. Numerous speech recognition APIs based on neural networks are available to enable rapid speech-to-text conversion in real-time applications and otherwise. With the evolution of deep learning algorithms, several advances in ASR have been made in the last decade. Initially, ASR systems were built using Hybrid Hidden Markov Models. End-to-end (E2E) ASR systems using sequence-to-sequence models have recently provided state-of-the-art results [19]. The most commonly used E2E ASR approaches are based on Connectionist Temporal Classification (CTC), Recurrent Neural Network Transducer (RNN-T), and attention-based encoder-decoder Transformer architectures.

However, in RNN-based approaches, the input sequences are read sequentially, causing trouble capturing long-range dependencies. The sequential processing of input inherently prevents the parallelization of computation. Self-attention-based Transformer architectures circumvent these shortcomings by processing a sequence as a whole, accounting for long-range dependencies. Applications like neural machine translation, language representation models, and other natural language processing tasks based on the Transformer architecture have produced promising results [32]. Similar successes have been observed in the domain of E2E ASR systems [8, 3, 26].

Inspired by the above, we propose a hardware accelerator for inference in a transformer-based end-to-end ASR system in this work. Although the transformer models have significantly improved accuracy, the required computations during an inference task are prohibitively high, which can be a significant bottleneck for ASR applications. For example, the particular transformer architecture (refer to Section 3.1) we have deployed requires 4 Giga floating-point operations to process a single input sequence, exerting the available compute resources. Fortunately, the transformer architecture is amenable to parallelization, allowing multiple computations can be performed simultaneously, allowing for more efficient processing. We exploit this to build an FPGA-based hardware accelerator for an E2E ASR system while

reducing the computational load and enabling the deployment of transformer-based ASR models in real-world scenarios.

A typical Transformer uses positional encoding, where each position of the input sequence is mapped to a vector that carries the positional information of the sequence. The recent developments led to various architectures where the positional encoding was replaced by Convolutional Neural Networks (CNNs) at the encoder and 1-D causal convolutions at the decoder layers. The convolutions at the decoder were eliminated in later developments [4]. We have used a similar E2E-trained model without positional encoding, which also eliminates the use of language and acoustic models individually.

The proposed hardware accelerator uses a systolic array-based structure for matrix multiplication as a building block. Eight systolic array structures, evenly distributed between the two Super Logic Regions (SLRs) of the FPGA, form the core compute fabric on which all encoder/decoder computations are routed. The host-side controller orchestrates all the encoder and decoder computations on the hardware accelerator without requiring reconfiguration between stages. The proposed architectural approach is flexible in two dimensions, as given below:

- It is possible to retarget the hardware accelerator to process different transformer networks with varying configurations, such as the number of encoders, decoders, and attention heads.
- Based on the availability of FPGA resources, we can appropriately determine the number and the dimensions of the systolic arrays for matrix multiplication, thus providing scalability on the parallelism front.

The accelerator design is realized using Vitis high-level synthesis (HLS) tool, parallelized using HLS-defined pragmas, and evaluated on an Alveo U50 FPGA card. The design demonstrates an average speed-up of 32$\times$ and 8.8$\times$ for a 32-bit single-precision floating-point model compared to an Intel Xeon processor and NVIDIA GeForce RTX GPU, respectively, for short to medium-length sequences. To the best of our knowledge, this is the first work that deploys an E2E ASR system on hardware.

1.2 Summary of Contributions

The main contributions of this thesis are explained in Chapter 4 and are mentioned briefly below :

- This work is the first implementation of an end-to-end Automatic Speech Recognition system on FPGAs.
- In this work, we identify the computational bottleneck in the Transfomer-based ASR model and describe a systolic-array-based Matrix Multiplication approach while ensuring maximum reuse of resources within various blocks of the model.
- We further compare various end-to-end architectures while overlapping the load and compute phases of the system, thus reducing the overall latency and increasing the throughput.

- We design the accelerator to exploit the inherent parallelism within the Transformer, with kernels running on both the Super Logic Regions of the device, while mitigating the inter-SLR communication. We compare our works with a CPU, GPU implementation, along with other related works.

1.3 Thesis Organization

The thesis is further divided into the following chapters :

- ***Chapter 2*** gives an introduction to NLP tasks like Automatic Speech Recognition (ASR) and its applications in various real-life scenarios. It talks about FPGAs as hardware devices, their advantages, and the use cases where they are highly effective. It also details the FPGA we have used and the fundamental process flow and optimizations in our context.
- ***Chapter 3*** explains the Transformer architecture. It presents the architecture's complexity and delineates why it is a challenge to implement on an FPGA.
- ***Chapter 4*** proposes a hardware architecture for a Transformer based model. It analyses each block in detail and finally discusses an end-to-end implementation for an ASR application.
- ***Chapter 5*** explains the experimental setup and discusses the results with a thorough analysis of our experiments.
- ***Chapter 6*** concludes the thesis and discusses the relevant work in this domain along with major takeaways from the project. It ends by discussing the future scope of this problem.
- ***Bibliography*** contains all the referenced papers used in this thesis.

Chapter 2

Automatic Speech Recognition on FPGAs

2.1 Automatic speech recognition

2.1.1 Introduction to Automatic Speech Recognition

Automatic Speech Recognition (ASR) is a key component in many technological advances, from virtual assistants to transcription and translation services. ASRs are computer programs designed to convert speech into text, allowing easier access to information and more efficient communication.

Speech is the most natural form of communication for humans, and ASRs allow us to interact with technology in a more natural way. With the increasing prevalence of smart speakers, smartphones, and other voice-activated devices, ASRs have become more important than ever. They enable us to control our devices, search for information, and communicate with others, all with the power of our voice.

ASRs have also revolutionized the field of accessibility for people with disabilities, particularly those who are deaf or have poor hearing. ASRs can transcribe spoken language into text, allowing for real-time communication in situations where it would otherwise be impossible. In addition to these practical applications, ASRs have also been the subject of extensive research in the field of machine learning and artificial intelligence. The development of ASRs has been a challenging task due to the complexity of speech as a signal. The ability to accurately transcribe speech requires overcoming issues such as background noise, varying speaking styles, and the lack of clear boundaries between individual words.

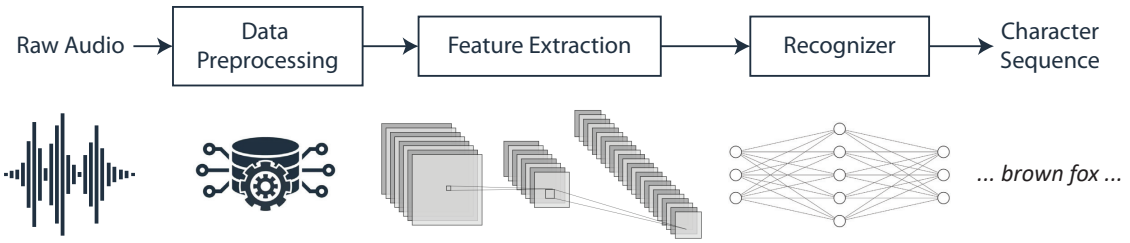

Figure 2.1 Structure of a typical ASR system

2.1.2 Evolution of Automatic Speech Recognition systems

The first ASR systems were based on pattern recognition techniques, which used templates to match incoming speech to stored reference patterns. These systems were limited in their accuracy and performance, as they required large amounts of memory and computational power to store and match reference patterns.

In the 1960s and 1970s, Hidden Markov Models (HMMs) were introduced as a new approach to ASR. HMMs allowed for more efficient modeling of speech by using statistical models to represent the temporal and spectral characteristics of speech. This allowed for better modeling of speech patterns, which improved ASR accuracy. Later, statistical ASR systems were developed, which used probabilistic models to estimate the likelihood of a given word or phrase given the input speech. These models included Gaussian Mixture Models (GMMs) and neural networks, which allowed for better modeling of speech patterns and improved ASR accuracy. However, the GMM-HMM-based ASR systems suffer from several limitations. [7] One of the main limitations is their inability to capture long-range dependencies in speech signals, which are critical for the accurate transcription of speech. Another limitation is their difficulty in handling the variability of speech signals, such as accent, speaking rate, and environmental noise.

In the 1990s, large-scale ASR systems were developed, which incorporated language modeling techniques to improve ASR accuracy which used statistical methods to estimate the probability of a given sequence of words based on their frequency and context. This allowed for more accurate recognition of spoken language, particularly in the presence of noise and other sources of interference.

2.1.3 Transfomer-based approach for ASR

Transformers are a type of neural network introduced in the natural language processing field by Vaswani et al. [32]. Transformers are designed to model long-range dependencies in sequential data and have achieved state-of-the-art performance in various natural language processing tasks such as machine translation, text classification, and language modeling. The core component of a Transformer is the self-attention mechanism, which allows the model to attend to different parts of the input sequence and weigh them according to their relevance to the current output. The self-attention mechanism enables Transformers to model long-range dependencies without the need for recurrent connections, which are computationally expensive and difficult to parallelize.

Transformers for ASR: In recent years, there has been a growing interest in using Transformers for ASR. Dong et al. [9], who introduced the Speech Transformer model, proposed the first use of Transformers for ASR. The Speech Transformer model uses a standard Transformer architecture with a modified input layer that converts the input speech signal into a sequence of feature vectors. The model is trained using Connectionist Temporal Classification (CTC) loss, which allows the model to learn the alignment between the input speech signal and the corresponding transcription. The Speech Transformer

model achieved state-of-the-art performance on the LibriSpeech dataset, which is a commonly used benchmark for ASR.

Subsequently, several variants of the Speech Transformer model have been proposed. For example, Liu et al. [30] proposed the RoFormer model, which uses a modified Transformer architecture with a relative position encoding scheme. The relative position encoding scheme allows the model to capture the relative position of different parts of the input sequence, which is important for accurately transcribing speech signals. The RoFormer model achieved state-of-the-art performance on several ASR datasets, including LibriSpeech and AISHELL-1.

Another variant of the Speech Transformer model is the Conformer model, which was proposed by Gulati et al. [13]. The Conformer model uses a modified Transformer architecture with a multi-head self-attention mechanism and a convolutional layer. The convolutional layer allows the model to capture local dependencies in the input speech signal, while the multi-head self-attention mechanism allows the model to capture global dependencies. The Conformer model achieved state-of-the-art performance on several ASR datasets, including the LibriSpeech dataset and the Switchboard dataset.

In addition to these models, several other variants of Transformers for ASR have been proposed, including the VGG-Transformer [37], which uses a combination of convolutional and Transformer layers.

The success of Transformers for ASR can be attributed to their ability to capture long-range dependencies in the input speech signal, as well as their parallelizable nature. A significant amount of research is being done to explore Transformer-based ASR models for handling noise and low-resource data, as they have shown state-of-the-art performance.

2.1.4 Standard ASR models used commercially

With the advent of speech technology-based applications, several popular commercial Automatic Speech Recognition systems are available in the market today. Here are some of the most notable ones:

Google Cloud Speech-to-Text: Google's ASR service uses advanced deep-learning techniques to recognize speech accurately in over 120 languages and dialects. It offers real-time transcription and can handle audio from multiple sources, such as microphones, audio files, and phone calls. It is a cloud-based service and can be easily integrated with other Google services and third-party applications. According to Google's documentation, the WER for the system ranges from 5-15% for general English speech. However, this can vary depending on the specific domain and language being spoken.

Amazon Transcribe: Amazon's ASR service is designed to transcribe audio files and live audio streams in real-time. It supports multiple languages, including English, Spanish, French, and Japanese, among others. It can also recognize specialized vocabularies, such as legal and medical terminology. Amazon claims that its ASR system has a WER of between 4-6% for conversational speech in English.

Microsoft Azure Speech to Text: Microsoft's ASR service offers real-time transcription of audio files and live audio streams. It supports multiple languages, including English, Spanish, Italian, and Chinese, etc. It also offers customizable models that can recognize specialized vocabularies and dialects. Azure Speech to Text can be easily integrated with other Microsoft services and third-party applications.

Microsoft claims that its ASR system has a WER of around 5% for the Switchboard benchmark, which is a standard benchmark for evaluating ASR performance in conversational speech.

IBM Watson Speech to Text: IBM's ASR service uses deep learning techniques to recognize speech accurately in multiple languages, including English, Spanish, French, German, Italian, and Japanese, among others. It can handle audio from various sources, such as microphones, audio files, and phone calls. It also offers customization options to train the model for specialized vocabularies and dialects. IBM Watson Speech to Text can be integrated with other IBM services and third-party applications. IBM's ASR system has reported a WER of between 6-12% for conversational speech in English.

These ASR systems are widely used in various industries, such as healthcare[25], finance, legal, and education[15], to automate speech recognition and transcription tasks. They offer high accuracy rates and can save time and effort in manual transcription and translation tasks. However, these models use high-end CPUs or hardware accelerators based on GPUs or FPGAs with hybrid models of HMM, CNN, and Transformers, as per their volume of data and requirements. For example, Google Cloud Speech-to-Text uses custom TPUs (Tensor Processing Units) for some aspects of its ASR pipeline.

2.2 Hardware Accelerators and FPGAs

2.2.1 Hardware accelerators

Hardware accelerators have become increasingly important in modern computing systems due to their ability to improve the performance and efficiency of various applications. Hardware accelerators provide a way to offload specific computations from the general-purpose processor, resulting in faster and more efficient processing. The evolution of hardware accelerators has been driven by the need to accelerate specific tasks that cannot be efficiently executed on traditional general-purpose processors. Initially, they were implemented as specialized processors for specific tasks, such as graphics processing or digital signal processing. However, the advent of Field-Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) has led to the development of programmable hardware accelerators. FPGAs are programmable logic devices that can be configured to implement specific hardware designs, making them ideal for implementing hardware accelerators. Conversely, GPUs are designed for parallel processing and are well-suited for accelerating certain types of computations, such as those involved in graphics rendering and machine learning.

The evolution of hardware accelerators has also been driven by the increasing demand for high-performance computing in various industries, such as finance[21], healthcare[31], and scientific research. Hardware accelerators have become essential to many computing systems, providing the necessary computational power to support these industries. Traditional general-purpose processors are unable to efficiently execute certain types of computations involved in machine learning to accelerate neural network training and inference, in scientific simulations, to accelerate the computation of complex mathematical models, and big data analytics[40].

They can also be optimized for specific applications, resulting in higher performance and efficiency than general-purpose processors. CPUs have a fetch-decode-execute cycle, a process by which instructions are fetched from memory, decoded into operations, and executed by the CPU. This process takes time and can limit the performance of the system. In contrast, FPGAs are designed to be highly parallelized and can execute multiple instructions simultaneously. This allows them to avoid the fetch-decode-execute cycle altogether and operate at much higher speeds than traditional CPUs.

Also, there has been a saturation in the performance improvements that can be achieved with general-purpose processors. To continue improving performance, there is a need for hardware specialization or the use of specialized hardware accelerators for specific tasks. The trend towards hardware specialization is evident in mobile processors, desktops, and data centers[17]. The needs for hardware accelerators also vary depending on the application.

2.2.2 Types of Hardware Accelerators

There are several types of hardware accelerators, each designed for specific tasks. Some of the most commonly used hardware accelerators are:

Field-Programmable Gate Arrays (FPGAs): FPGAs are programmable logic devices that can be configured to implement specific hardware designs. They are well-suited for implementing hardware accelerators due to their high configurability and low power consumption.

Graphics Processing Units (GPUs): GPUs are designed for parallel processing and are well-suited for accelerating certain types of computations, such as those involved in graphics rendering and machine learning[24].

Digital Signal Processors (DSPs): DSPs are specialized processors designed for processing digital signals, such as those involved in audio and video processing[33].

Application-Specific Integrated Circuits (ASICs): ASICs are custom-designed integrated circuits that are optimized for specific tasks. They are well-suited for implementing hardware accelerators for applications that require high performance and efficiency[23].

The wide range of uses for FPGA-based accelerators in commercial and military aircraft, automotive computers, cryptography devices [2], GPS, image processing, Internet of Things (IoT) devices [14], medical imaging, satellites [12], security systems, smartphones, spacecraft, supercomputers [11], unmanned vehicles, voice recognition systems, wireless devices, etc.

2.2.3 Field Programmable Gate Arrays

An FPGA is a semiconductor device containing programmable logic elements and programmable interconnect but no instruction fetch, i.e., they do not have a Program Counter. FPGAs are reprogrammable hardware devices that are used to run a variety of computing applications. They are used to accelerate any computing task that runs slow on CPUs or consumes large amounts of power when run on GPUs.

With their reconfigurability, they fill the gap between software and hardware, achieving potentially much higher performance than software while maintaining a higher level of flexibility than hardware [6]. It can be used by implementing all the application functionalities in hardware and covers all data paths from input to output. The advantage of doing this lies in the fact that the functionality of the hardware can be changed just by loading a different configuration file from the host instead of configuring the hardware with a new physical circuit.

An FPGA consists of various components that work together to perform specific functions. The main components of an FPGA include the following:

- Logic Elements: These are the basic building blocks of an FPGA and are used to implement combinational and sequential logic circuits. Each Logic Element consists of a lookup table (LUT), a flip-flop (FF), and a multiplexer.
- Configurable Interconnect: This component is responsible for routing signals between different logic elements in the FPGA. The interconnect is configurable that allows designers to create custom routing paths optimized for their specific application.
- Input/Output Blocks (IOBs): These blocks are used to interface the FPGA with the external world. IOBs are responsible for receiving input signals, transmitting output signals, and implementing various input/output standards such as LVDS, LVPECL, and SSTL.
- Block RAM (BRAM): This component is used to implement random-access memory (RAM) in an FPGA. Block RAM is configurable and can be used to implement any size and depth of memory as required by the design.
- Digital Signal Processing (DSP) Blocks: DSPs are specialized components that are used to perform digital signal processing operations such as multiplication, addition, and filtering. DSP blocks typically consist of multipliers, adders, and registers.
- Clock Management: This component is responsible for managing the clock signals used in the FPGA design. The clock management circuitry includes phase-locked loops (PLLs) and delay-locked loops (DLLs) that are used to generate and distribute clock signals across the FPGA.

2.2.4 Alveo U-50 Data Accelerator Card

We use an Alveo U50 Data Accelerator for our implementation. It features a Xilinx UltraScale+ FPGA, which provides high-speed connectivity, high bandwidth memory, and a low-latency network interface.

The U50 accelerator card consists of 8GB of HBM2 memory, which provides high bandwidth and low latency access to the FPGA fabric. It also supports up to 100 Gb/s networking with built-in Ethernet connectivity. The card also includes a PCIe interface, which enables it to be easily integrated into existing server infrastructures.

Figure 2.2 Alveo U-50 FPGA [1].

The Xilinx UltraScale+ FPGA on the U50 accelerator card provides up to 1743K registers, 5,952 DSP slices, and 2,688 BRAM blocks. These resources allow the U50 to accelerate a wide range of applications, including machine learning, data analytics, and image or speech processing.

Super Logic Regions: The device consists of two Super Logic Regions (SLRs) and has the primary resources like BRAM, DSP, FF, and LUT distributed approximately equally between the two SLRs. SLRs were introduced in FPGA's to group together the related logic resources that are commonly used in a particular application. This grouping allows the resources to be placed close to each other and connected by high-speed interconnects that are optimized for that specific application. SLRs can also be used for partitioning the design to facilitate easier design and debugging. Any complex design can be partitioned into smaller regions, each with its own SLR, to simplify the design process and reduce the debugging time. We try to optimize the logic distribution and scheduling of operations to ensure minimal inter-SLR communication. Fig. 2.3 shows the floor-plan of an XCU-50 device with two Super Logic Regions.

The Alveo U-50 is also characterized by High Bandwidth Memory (HBM). HBM is designed to solve the bandwidth bottleneck that exists in traditional memory architectures, where the memory bandwidth is limited by the number of pins available to connect the memory to the processor. With HBM, the memory is stacked on top of the device, providing a high-speed communication path between the memory and the processor.

High Bandwidth Memory: HBM provides several advantages over traditional memory architectures, including:

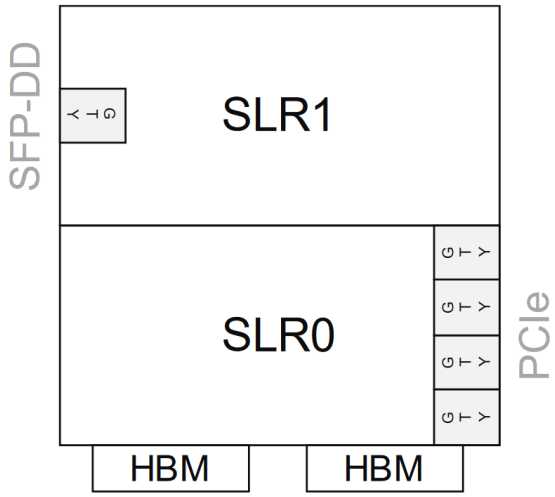

Figure 2.3 Floor plan of Alveo U-50 FPGA [1]

.

- Higher bandwidth: HBM provides much higher memory bandwidth than traditional memory architectures, which can help improve the performance of memory-intensive applications.
- Lower power consumption: HBM consumes less power than traditional memory architectures, making it an ideal choice for energy-efficient systems.
- Smaller form factor: HBM has a smaller form factor than traditional memory architectures, which can help reduce the size of the overall system.
- Higher memory density: HBM has a higher memory density than traditional memory architectures, allowing for more memory to be packed into a smaller space.

HBM Communication with both SLRs: Fig. 2.3 shows that the HBM on the Alveo U50 accelerator card is connected to only one Super Logic Region on the FPGA. However, data can still be communicated between the HBM and other SLRs on the FPGA by using the Inter-SLR Communication (ISC) interface.

It allows SLRs on the FPGA to communicate with each other by providing a high-bandwidth and low-latency interconnect between the SLRs. The ISC interface uses a shared memory architecture, with each SLR having access to a portion of the shared memory. This allows data to be easily shared between SLRs.

To communicate with the HBM from another SLR on the FPGA, the kernel running on the Alveo U50 card can use the ISC interface to access the shared memory region containing the HBM data. The kernel can then perform computations on the data as required. The ISC interface on the Alveo U50 card is implemented using the AXI Stream interface, which allows data to be transferred between SLRs in

a streaming fashion. Thus the kernel running on the Alveo U50 card performs computations on data located in multiple SLRs, which can be important for many data-intensive workloads.

Peripheral Component Interconnect Express: PCIe is a high-speed serial interface that is used to connect the Alveo U50 accelerator card to the host system's CPU and memory. It provides a high-bandwidth, low-latency connection between the host and the Alveo U50 card, enabling the card to access the host's memory.

The Alveo U50 card appears to the host system as a standard PCIe device, and the host system communicates with the card using standard PCIe protocols. The host system can initiate DMA transfers to and from the Alveo U50 card, allowing data to be transferred between the host's memory and the HBM on the Alveo U50 card.

2.2.5 Vitis HLS

Vitis HLS (High-Level Synthesis) is a tool that allows software developers to design hardware accelerators for FPGA-based systems. Vitis HLS uses high-level programming languages, such as C/C++, and converts them into RTL (register-transfer level) code. This RTL code can then be synthesized into a bitstream that can be loaded onto an FPGA to implement the hardware accelerator.

Vitis HLS allows for an iterative design process, where the designer can quickly modify the high-level code, run simulations to verify functionality, and then generate RTL code for synthesis.

One of the key features of Vitis HLS is its support for pipelining and parallelism. Pipelining allows for the processing of multiple instructions at the same time, which can increase throughput and reduce latency. Parallelism allows multiple operations to be performed simultaneously, further increasing performance. Vitis HLS includes tools to automatically detect pipeline and parallelism opportunities in the code and generate the necessary hardware structures to implement them.

Another key feature of Vitis HLS is its support for interface synthesis. Interfaces allow for communication between the hardware accelerator and the rest of the system. Vitis HLS supports several different interface types, including AXI (Advanced eXtensible Interface) and FIFO (First-In, First-Out). Vitis HLS can automatically generate the necessary interface logic based on the specified interface type. Vitis HLS includes a range of optimization options to further improve performance and reduce resource utilization. These optimization options include loop unrolling, resource sharing, and loop pipelining.

2.2.6 Pragmas in Vitis HLS

Vitis Pragmas are compiler directives that allow the user to control the hardware implementation of a software function when using the Xilinx Vitis HLS tool. Vitis Pragmas provide a way to specify various optimization options and hardware configurations that affect the performance and resource utilization of the generated hardware design.

There are many Vitis Pragmas available, each with its own specific purpose. Here are some commonly used Vitis Pragmas and their functions:

- PIPELINE - This pragma is used to pipeline a loop or function. It allows the loop or function to execute in parallel with other operations, improving performance.
- UNROLL - This pragma is used to unroll a loop. Unrolling a loop means that the loop body is replicated multiple times, allowing multiple iterations to be executed in parallel. This can improve performance by reducing the loop overhead.
- ARRAY_PARTITION - This pragma is used to partition an array into smaller pieces. Partitioning an array can help reduce the number of memory accesses required, improving performance.
- INTERFACE - This pragma is used to specify the interface between the hardware function and the rest of the system. This pragma allows the user to specify the data types, data widths, and other interface parameters.
- DATA_PACK - This pragma is used to pack multiple data elements into a single memory location. Data packing can help reduce the number of memory accesses required, improving performance.
- DATAFLOW - This pragma is used to specify that a function is dataflow-oriented. Dataflow-oriented functions are executed as soon as all of their input data is available, allowing them to execute in parallel with other operations.

We have several pragmas in various regions of the code for the implementation of an end-to-end transformer-based ASR system. Some of them are discussed below.

We use the pragma 'Loop unroll' in the matrix multiplications to implement multiple sections of a loop in parallel. We have the flexibility to define the number of parallel sections by defining the unroll factor.

We use the Pragma 'DATA FLOW' to schedule the Value matrix computation (multiplication with a weight matrix), along with scaling and softmax of the attention-scores. Thus we reduce the overall latency of each attention head in the Multi headed attention block.

The pragma 'ARRAY_PARTITION' helps in partitioning large arrays like weight matrices or Query/Key/Value scores. The 'PIPELINE' pragma allows overlapping of read and compute phases as required in the end-to-end architecture.

2.2.7 Host-side Process Flow

The process flow on the host side is executed using OpenCL in the following stages:

- The host code initializes the OpenCL platform and creates a context for the accelerator card. The context specifies the devices that will be used for computation. Then the host code compiles the OpenCL program that will run on the accelerator card. It contains the kernels to be executed on the FPGA. It also specifies which SLR each kernel will execute on. The program is compiled to generate a binary file that will be loaded onto the device.

- The host code allocates memory on the device for input and output data. This is typically done using OpenCL memory allocation APIs, and the host code transfers input data from the host to the device using OpenCL memory transfer APIs. The data is then made available for the kernels to access during execution.
- The host code launches the kernels on the accelerator card using OpenCL command queue APIs, The host code specifies the SLR on which each kernel should execute using OpenCL device fission APIs. The kernels execute in parallel on the FPGA fabric, processing the input data.
- Once the kernels have completed execution, the host code retrieves the output data from the device using OpenCL memory transfer APIs and makes the output data available for the host to process.

The intermediate processing of this data on the kernels from the host is executed on the FPGA-based accelerator to provide the final output data to the host. This accelerator architecture will be discussed in Chapter 4.

Chapter 3

Automatic Speech Recognition Model Architecture and Analysis

3.1 End-to-end Automatic Speech Recognition Model

The end-to-end ASR model is trained and implemented using the End-to-End Speech Processing toolkit (ESPnet), an open-source platform for speech recognition [36]. The character-level-based E2E speech processing includes data preparation, feature extraction, and attention-based encoder-decoder architecture. The feature extraction process uses a simple Mel filter-bank technique. Here, the signal is passed through a pre-emphasis filter, which emphasizes (i.e., increases the amplitude of) the high-frequency components of the signal and de-emphasizes (i.e., decreases the amplitude of) the low-frequency components. The purpose of pre-emphasis is to improve the signal-to-noise ratio and to compensate for the high-frequency energy that is lost during recording or transmission. The resultant signal is split into short frames of 25ms and passed through a window function. We perform a Short-Time Fourier Transform (STFT) by breaking down a signal into short-time segments, called frames, and then performing a Fourier Transform on each frame. This results in a matrix of complex numbers, where each row corresponds to a frequency band and each column corresponds to a time frame. The magnitude of each complex number represents the amplitude of the corresponding frequency band at that particular time. We then apply triangular filters of 80 dimensions to obtain the filter banks. Triangular filters are commonly used to construct filter banks, as they provide a good approximation of the human auditory system's frequency response. The features generated are passed through a 2D convolutional layer, followed by a max-pool layer. This is followed by the encoder-decoder-based Transformer architecture, which is detailed below.

3.2 ESPnet tool kit

ESPnet is an open-source toolkit for end-to-end speech processing. It was developed by the Speech Processing Laboratory at Nagoya University and is maintained by a community of researchers and engineers.

ESPnet provides a range of features for speech processing, including automatic speech recognition (ASR), text-to-speech (TTS), and speaker recognition. It is designed to be modular and customizable, allowing researchers and developers to easily modify and extend the toolkit to suit their specific needs. One of the key features of ESPnet is its use of neural network models for speech processing. These models are trained using large amounts of speech and text data, allowing them to accurately transcribe speech into text or synthesize speech from text.

ESPnet also includes a range of pre-trained models for ASR and TTS that can be used out of the box or fine-tuned for specific tasks. The toolkit supports a range of languages, including English, Mandarin, and Japanese. ESPnet is a powerful and flexible toolkit for end-to-end speech processing that is widely used in research and industry. Its modular design and support for neural network models enable the building of custom speech-processing applications.

3.3 Libri Seech Dataset

The model has been trained on the LibriSpeech dataset [27], a large-scale corpus of read English speech data created using a subset of the publicly available LibriVox project containing free audiobooks that are read by volunteers. The LibriSpeech dataset consists of over 1,000 hours of speech data from a diverse range of speakers, including both male and female speakers of different ages and accents.

The dataset is provided in the form of raw waveform files in 16-bit PCM format with a 16 kHz sampling rate. The speech data is split into segments based on sentence boundaries, with each segment stored as a separate file. The dataset also includes accompanying text transcripts in plain text format. The LibriSpeech dataset is widely used in research on automatic speech recognition (ASR), speech-to-text translation, and other speech-related tasks and it is effective for training deep neural networks and has been used as a benchmark dataset for evaluating ASR systems.

3.4 Transormer Model Description

The Transformer model is a deep learning architecture used primarily in natural language processing (NLP). It has become a popular alternative to recurrent neural networks (RNNs) for NLP tasks, particularly in the context of Automatic Speech Recognition [13], [10], due to its parallel processing capabilities, compared to the sequential processing of RNN-based models.

At its core, the Transformer model relies on a self-attention mechanism, which allows it to process input sequences in parallel rather than sequentially. This is achieved through the use of multiple layers of attention-based encoders and decoders, which operate on the input sequence and generate an output sequence. The self-attention mechanism in the Transformer model allows it to capture long-range dependencies between elements in the input sequence, which is particularly useful in NLP tasks, such as ASR where the meaning of a word or phrase may depend on other words or phrases in the sentence.

The Transformer architecture consists of a sequence of encoders followed by decoders. The encoder is primarily responsible for mapping the input, a feature vector here, into a sequence representation, further fed to the decoder. The posterior probability of the label is predicted by applying a fully-connected layer and a softmax distribution over the decoder stack output. The Transformer model we deployed consists of an encoder stack consisting of 12 identical encoders and a decoder stack consisting of 6 identical decoders. It is as shown in Fig. 3.1.

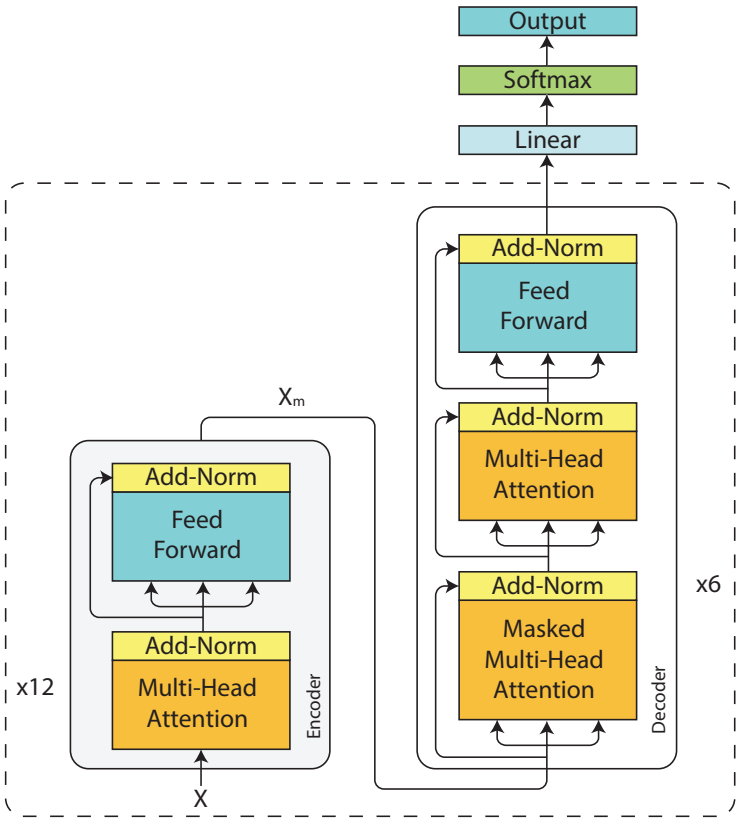

Figure 3.1 Transformer architecture

Each encoder contains a Multi-Head Attention block (MHA) and a Feed-Forward Network block (FFN), each followed by an Add-Norm layer. Each decoder in the decoder stack contains a Masked Multi-Head Attention (M-MHA), Multi-Head Attention (MHA), and Feed-Forward Network (FFN), each followed by an Add-Norm layer. The MHA block comprises eight attention heads (h). The input embedding (X) is passed through linear sub-layers denoted by weights ($W_{Q/K/V}$) and biases ($B_{Q/K/V}$), obtained during training. It results in three projections of the input sequence, namely, Query (Q), Key (K), and Value (V). The product of the matrices Q and K^T is a correlation matrix of words, with their attention scores provided to the scaling (Sc) and softmax (Sm) function. The result is multiplied by the Value matrix, resulting in the attention score. The attention scores are used to compute a weighted sum of the input sequence, based on how relevant each position in the input sequence is to each position

in the output sequence. We add a look-ahead mask at the M-MHA layer of the decoder. The look-ahead mask is a binary mask that ensures that the decoder only attends to tokens that have already been generated and produces high-quality output sequences that are conditioned on the input sequence and the previously generated tokens.

$$Attention(Q, K, V) = Softmax(\frac{QK^T}{\sqrt{d_K}})V \tag{3.1}$$

The embedding size (d_{model}) used in the model is 512. The embedding size determines the number of features that are extracted from each frame of the input audio signal. These features are then transformed into a lower-dimensional representation to be processed by the transformer encoder. A larger embedding size typically allows for the model to learn more complex patterns in the input signal but also requires more computational resources. The scaling parameter is obtained as the embedding size divided by the number of attention heads, i.e., $d_k = d_{model}/h = 64$. The result from each attention head is concatenated and passed through another linear layer (W_A, B_A), followed by an Add-Norm block as shown in Fig. 3.2.

$$MHA(X) = (Concat(head_1, .., head_8)W_A) + B_A \tag{3.2}$$

The FFN consists of two linear transformations with a ReLU activation function between the layers, as shown in Fig. 3.3.

$$FFN(X_m) = ReLU(W_{1F}(X_m) + B_{1F})W_{2F} + B_{2F} \tag{3.3}$$

Here, X_m is the output of the MHA-Add-Norm block. In a traditional transformer architecture, the decoder input at each time step consists of the output from the previous time step and the multi-head attention (MHA) output. This means that the decoder input at each time step depends on the previous time step output, which can limit the model's ability to generalize to new input sequences. In contrast, E2E transformer-based ASR models eliminate the previous time step output of the decoder as input. The model receives input solely from the MHA output at each time step. This approach helps to increase the model's generalization capabilities by reducing the dependency on the previous time step output.

W_{1F}, B_{1F}, W_{2F}, and B_{1F} correspond to the weight and bias matrices. After the MHA and FFN layers process the input, the output is added to the input of the layer and then normalized. This helps to prevent the gradients from exploding or vanishing during the training process, as well as improve the stability of the model. The Add function is responsible for passing the input forward in the network, while layer normalization ensures that the output has a mean of zero and a standard deviation of one.

The Add-Norm layer (N) can be calculated as $N = w \cdot H + b$, where w and b are weight and bias parameters.

$$H = \frac{X - \mu}{\sigma},\ \mu = \frac{1}{D}\sum_{i=1}^{d} x_i,\ \sigma^2 = \frac{1}{D}\sum_{i=1}^{D}(x_i - \mu)^2 \tag{3.4}$$

where X = $x_1, x_2, ..x_D$ is the sum of MHA/FFN output and Add-Norm input.

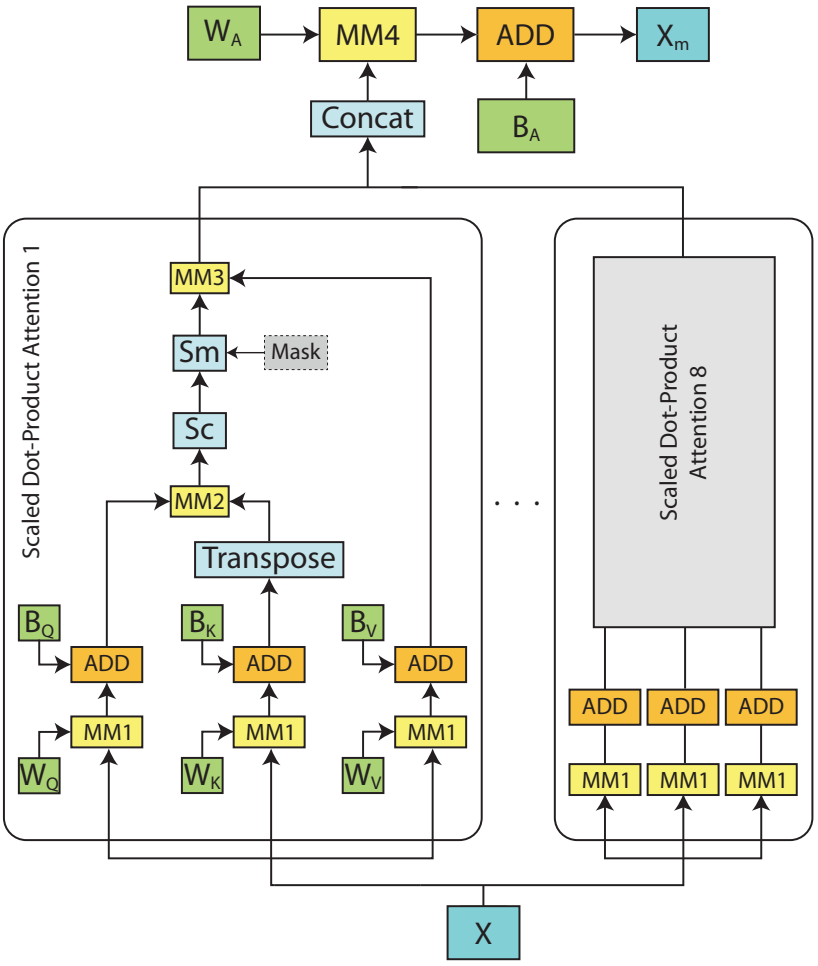

Figure 3.2 Block-level implementation of the MHA

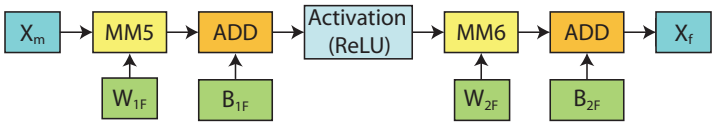

Figure 3.3 Block-level implementation of the FFN

The following chapter details the MHA and FFN implementation on the hardware and the controller orchestrating the encoder and decoder stages on these hardware structures while optimizing the accelerator architecture for resources and latency.

Chapter 4

Proposed Hardware Implementation of the Algorithm

In Chapter 3, we explored the transformer architecture in detail. In this chapter, we discuss in detail the proposed hardware architecture of our accelerator. Section 4.4 addresses the primary constraints of the model, which involve optimizing resource utilization and resource reuse while considering the model's size. In the later sections, we present our end-to-end architecture and the resource scheduling structure of the accelerator.

4.1 Experimental setup

Recall that the E2E ASR Transformer architecture deployed consists of twelve encoders followed by six decoders. Each encoder/decoder is scheduled for computation sequentially on the hardware accelerator by the host. Thus the proposed approach works for any sequence-to-sequence translation Transformer architecture consisting of an arbitrary number of encoder/decoder layers. The ASR application being run on the host offloads a computationally expensive task, the Transformer architecture here, onto the hardware kernel. The host program writes the data required by a kernel into its global memory through the PCIe (Gen 3×16) interface, which has a performance rate of 8 GigaTransfers/second. It sets up the kernel with its input parameters and triggers the execution. The kernel loads data from global memory to the on-chip BRAM and performs the computations. It writes back the data to global memory, which is read to the host. We use the M-AXI interface for data transfer to the kernel. This is demonstrated in Fig. 5.3

4.2 Operational intensity

Operational intensity is a metric used to quantify the amount of computation performed relative to the amount of memory accessed during the execution of an algorithm or program. It is calculated as the ratio of the number of floating-point operations to the amount of data transferred from memory. The operational intensity of this architecture is calculated as the number of Floating Point Operations (FLOPS) by total floating point data transferred in bytes (B). The operational intensity of the Transformer ar-

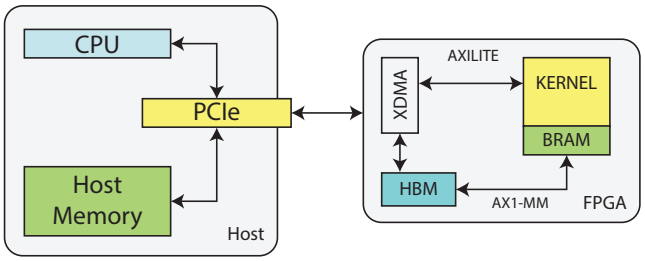

Figure 4.1 Overview of the accelerator setup

chitecture for a given sequence length is approximately 0.25 FLOPS/B. The Matrix Multiplication is an integral block demonstrating the architecture's highest utilization, latency, and operational intensity, emphasizing the requirement to optimize the matrix multiplication block. TABLE 4.1 tabulates the weight matrices read at various stages of computation in the Transformer. W and B correspond to weights and biases. L_N denotes Layer-normalization.

Table 4.1 Weight matrices read for an encoder-decoder stack

Number of matrices	***Weight matrix***	***Matrix dimensions***
576	$W_{Q/K/V}$	512×64
576	$B_{Q/K/V}$	1×64
24	W_A	512×512
24	B_A	1×512
84	L_N	1×512
18	W_1F	512×2048
18	B_1F	1×2048
18	W_2F	2048×512
18	B_2F	1×512

4.3 Paralleism scheme

We now discuss the solutions to the parallelism structure of the Transformer based model implemented on hardware.

An encoder consists of one MHA block followed by an FFN block. A decoder consists of two MHA (masked) blocks in series followed by an FFN block. Each MHA block has eight attention heads computed in parallel with four attention heads on each of the available two SLR regions on the FPGA. Each attention head has five matrix multiplication operations: three MM_1 operations corresponding to Q, K, V linear projection matrices; one MM_2 operation for QK^T; and one MM_3 operation for $Softmax(\frac{QK^T}{\sqrt{d_K}})V$. MM_4 is the last matrix multiplication operation outside the attention blocks within MHA. Here, the MM_1 operations within each attention head are executed sequentially due to a lack of

hardware resources. The FFN block has two matrix multiplication operations, MM_5, followed by MM_6. Thus each encoder and decoder computation consists of 8 and 14 matrix multiplications. Table 4.2 provides the dimensions of all these matrix multiplications.

Table 4.2 Dimensions of the Matrix Multiplication operations where s is the sequence length.

MatMul	***Input 1***	***Input 2***	***Output***	***Figure***
MM_1	s × 512	512 × 64	s × 64	Fig. 4.3
MM_2	s × 64	64 × s	s × s	Fig. 4.4
MM_3	s × s	s × 64	s × 64	Fig. 4.4
MM_4	s × 512	512 × 512	s × 512	Fig. 4.5
MM_5	s × 512	512 × 2048	s × 2048	Fig. 4.6
MM_6	s × 2048	2048 × 512	s × 512	Fig. 4.7

4.4 Orchestrating Matrix Computations

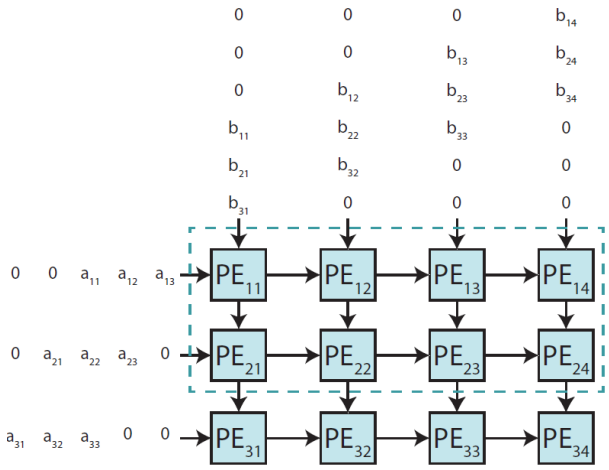

Figure 4.2 Systolic array for multiplication of two matrices of dimensions 3×3 and 3×4.

The block diagram of the encoder and decoder layers from Fig. 3.2 and Fig. 3.3 suggest that the general Matrix Multiplications (MM) take up the largest portion of the computation. As discussed in Section 4.2, we emphasize optimizing the matrix multiplication operation. To reduce the time complexity of this operation, we use a Systolic Array-based (SA) design, which is widely used in FPGA implementations.

A systolic array is a system that computes and passes data through a grid of processing elements. The inputs from the matrices flow through this structure rhythmically. The sequential implementation

of a general Matrix Multiplication operation has a time complexity of O(n3), for n x n matrices, while the SA-based design reduces it to O(n).

However, the parallel calculation of every element is limited by resources, primarily DSPs and LUTs. Given the resource availability, we use *partially unrolled systolic arrays* (PSAs). We loop-unroll the systolic array structure, thereby increasing the latency by at least $\sim 16\times$ while significantly reducing the DSP and LUT utilization.

Algorithm 1 Partially unrolled systolic array where $N_1 : s$, $N_2 : 64$, $N_3 : 64$

Input 1: Matrix $(N_1 \times N_3)$, Input 2: Matrix $(N_3 \times N_2)$
Output: Matrix $(N_1 \times N_2)$

```
for k = 0 to N3
    for i = 0 to N1, i+=2
        for j = 0 to N2, j+=N2

            a(i, j, k) = a (i, j-1, k);
            b(i, j, k) = b(i-1, j, k);
            c(i, j, k) = c(i, j, k-1) + a(i, j, k) * b(i, j, k);

            a(i+1, j, k) = a (i+1, j-1, k);
            b(i+1, j, k) = b(i, j, k);
            c(i+1, j, k) = c(i+1, j, k-1) + a(i+1, j, k) * b(i+1, j, k);
             .
             .
            a(i, j+N2-1, k) = a (i, j+N2-2, k);
            b(i, j+N2-1, k) = b(i-1, j+N2-1, k);
            c(i, j+N2-1, k) = c(i, j+N2-1, k-1) + a(i, j+N2-1, k) * b(i, j+N2-1, k);

            a(i+1, j+N2-1, k) = a (i+1, j+N2-2, k);
            b(i+1, j+N2-1, k) = b(i, j+N2-1, k);
            c(i+1, j+N2-1, k) = c(i+1, j+N2-1, k-1) + a(i+1, j+N2-1 , k) * b(i+1, j+N2-1, k);

        end
    end
end
```

All the matrix multiplications from MM_1 to MM_6 are routed on these PSAs. Fig. 4.2 shows the standard systolic array structure for matrix multiplication. The dimensions of the input matrices used in the example illustration are 3×3 and 3×4. In general, the product of two matrices $A_{l\times m}$ and $B_{m\times n}$ can be performed in $\Theta(m)$ time using a full $l \times n$ dimensional systolic array requiring ln Multiply-And-Accumulate (MAC) structures. We can trade off parallelism with area by computing the product matrix b rows or columns simultaneously. Note that the i^{th} row of the product matrix C is a linear combination of the m rows of matrix B with m coefficients coming from the i^{th} row of matrix A. This

observation allows us to compute b product rows in parallel, thus reducing the dimensionality of the systolic array to $b \times n$. For example, in Fig. 4.2, we have the first two rows of the systolic array to save on area and render the output product matrix two rows at a time. Similarly, we can compute b product columns in parallel with a $l \times b$ systolic array. However, with large dimensions of l and n, we may not be able to afford hardware resources, even with the reduced dimensionality of systolic arrays. We encounter this scenario while performing certain matrix multiplications in our Transformer architecture. We handle this by block-stripping the matrix multiplication operation, which is explained subsequently in this section. Altogether, we use eight PSAs with four PSAs per SLR region. The number of PSAs and their dimensionality determines the overall parallelism in computations and the hardware resources utilized, such as LUTs, DSPs, etc. We arrived at the 2×64 dimension by experimentally evaluating different PSA configurations for latency and area.

$\mathbf{MM_1}$ Computation: From Table 4.2, we can see the dimensions of the input matrices are $s \times 512$ and 512×64, and the output matrix dimensions are $s \times 64$. MM_1 is implemented using a PSA block iteratively eight times, as demonstrated in Fig. 4.3. Input1 and Input2 are partitioned into eight stripes column-wise and row-wise, respectively. The dimensions of the column and row striped block matrices are $s \times 64$ and 64×64. Each pairwise product generates a partial product matrix of dimensions $s \times 64$. Eight partial product matrices are added using an adder pipelined with the PSA to form the final output matrix. Pipelining the adder reduces the latency from $8t_{PSA} + 7t_{ADD}$ to $8t_{PSA} + t_{ADD}$. Fig. 4.3 also shows how the pipelined computations are scheduled. The three MM_1 operations within an attention head are scheduled for computation sequentially on a single PSA structure. The MM_1 operations from the eight attention heads are processed in parallel by the eight PSA structures.

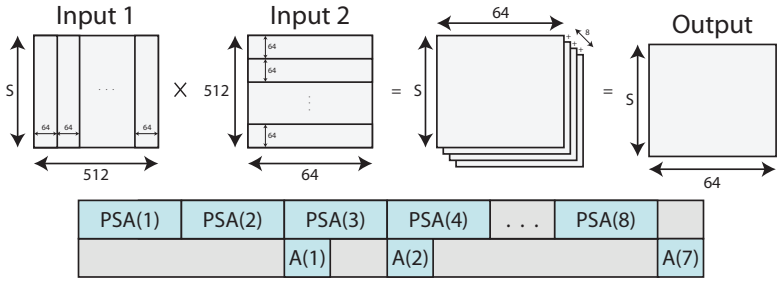

Figure 4.3 MM_1 block with Input1, Input2 and Output

$\mathbf{MM_2}$ and $\mathbf{MM_3}$ Computations: Since the dimensions of the input matrices for MM_2 and MM_3 are small (refer Table 4.2), we route the computations on a PSA block using appropriately padded input. Fig. 4.4 shows how the input matrices are padded. Although it contributes to a higher latency for shorter sequences, reusing a PSA block reduces the hardware footprint. It is also possible to halt the computation and read the output from the PSA structure by adding appropriate control signals to the PSA structure. However, we did not do this optimization in our implementation.

$\mathbf{MM_4}$ Computation: Table 4.2 suggests that the dimensions of the input matrices in MM_4, MM_5 and MM_6 are much higher when compared to MM_1, MM_2 and MM_3. However, a single instance of these

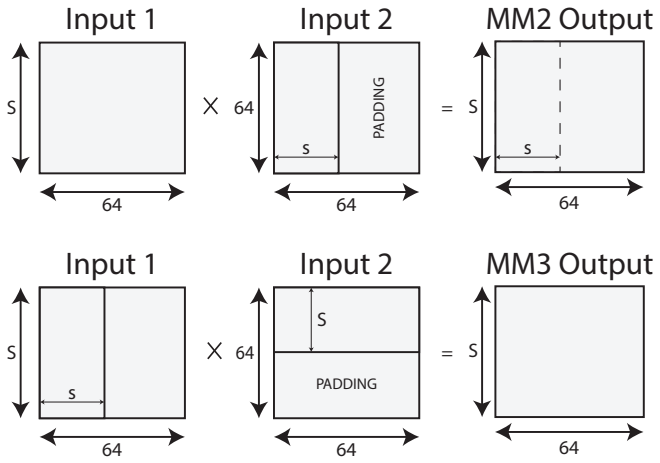

Figure 4.4 MM_2 (top) and MM_3 (bottom) blocks with Input1, Input2, and Output

operations is active for scheduling at a time. We exploit this observation to route these large matrix multiplication operations across the two SLRs on all eight PSA blocks.

The concatenated output from the eight attention heads and along with weight matrix W_A, are the inputs to MM_4. This computation is performed by partitioning the first input matrix into eight-column stripes of dimension $s \times 64$ and the weight matrix into eight-row stripes of dimension 64×512. This generates eight partial products whose accumulation gives the final product matrix of dimension $s \times 512$. The first four column stripes come from the four attention blocks in the SLR0 region of the FPGA. Similarly, the second four column stripes come from the rest of the four attention heads in the SLR1 region. Thus the whole MM4 computation is distributed on the eight PSAs from the two SLR regions. Figure 4.5 shows matrix decomposition, concatenation, multiplication, and accumulation (with a pipelined adder) steps within MM_4.

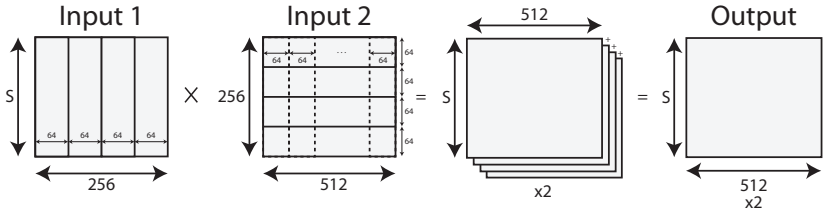

Figure 4.5 MM_4 block with Input1, Input2, and Output

$\mathbf{MM_5}$ Computation: The inputs to the MM_5 block are $s \times 512$ and 512×2048. However, each SLR receives a partitioned 512×1024 weight matrix. The first input matrix is divided into two $s \times 256$ dimensional matrices. The second input matrix is divided into four 256×512 matrices. Within an SLR region, the two partitioned $s \times 512$ matrices of the bigger $s \times 1024$ matrix are computed independently using a scheme similar to MM_4. Fig. 4.6 summarizes the overall scheme of computation. The resultant product matrix $s \times 2048$ is distributed across the 2 SLR regions with columns split equally.

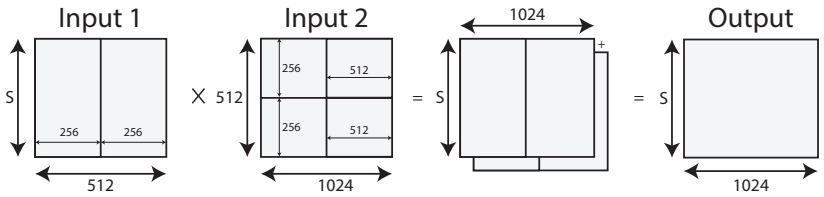

Figure 4.6 MM_5 block with Input1, Input2 and Output

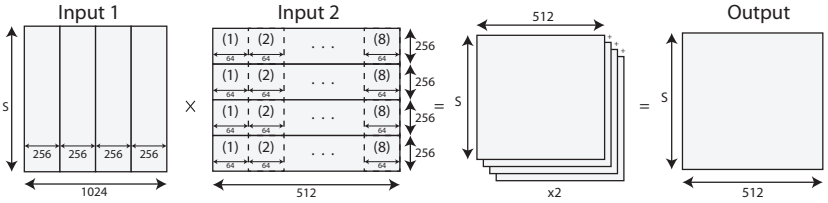

Figure 4.7 MM_6 block with Input1, Input2 and Output

$\mathbf{MM_6}$ Computation: Similarly, MM_6 computations can be scheduled based on MM_4. The inputs to the MM_6 block are $s \times 2048$ and 2048×512. However, each SLR receives a partitioned 1024×512 weight matrix. The first input matrix is divided into four $s \times 256$ dimensional matrices. The second input matrix is divided into four 256×512 matrices. Within an SLR region, the four matrix multiplications of the partitioned inputs are performed using a scheme of operation similar to MM_4. The final output matrix of dimension $s \times 512$ is the output of the FFN block. Fig. 4.7 summarizes the overall scheme of computations.

4.5 Computation and Communication Overlap

Architecture-1 (A1): The weight matrices are read onto the kernel memory from the global memory (HBM) in burst mode. For a given input sequence, the accelerator reads a set of weights twelve times for an encoder stack and six times for a decoder stack from the global memory. The last decoder writes back a $s \times 512$ matrix to the global memory. Thus, the architecture primarily involves load (LW_i) and compute (C_i) phases. Every LW_i involves loading weight matrices of one Encoder/Decoder, and C_i corresponds to one encoder/decoder execution across both the SLRs. Architecture A1 follows a naive sequential execution of load and compute phases of every encoder/decoder, i.e., a load of the first encoder (LW_1), followed by Compute of the first encoder (C_1) and so on until the twelfth compute (C_{12}). The decoder follows a similar architecture.

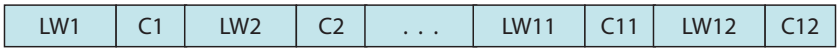

Figure 4.8 Architecture A1 for Encoder stack

Architecture-2 (A2): The time taken to load the weights onto the BRAM is higher than the computation time for short input sequence lengths. We cannot achieve task-level parallelism in the compute phase due to the dependency for every compute block with the previous block. Hence, to reduce the effect of the load time on the latency, we use a well-established technique for task-pipelining, where the load and compute operations happen simultaneously. C_i is executed after LW_i, in parallel to $LWi + 1$ which acts as a buffer. This is elucidated for an encoder stack in Fig. 4.9.

Figure 4.9 Architecture A2 for Encoder stack.

Architecture-3 (A3): The architecture A2 still suffers from stalls in the compute phase. We propose an architecture for optimal latency without further expending any resources by performing overlapping load operations from different channels of the HBM memory. The LW_{i+2} is initiated immediately after C_i is computed. The blocks are tiled, ensuring the stall at compute phase is reduced from $LW_i - C_i$ to $(LW_i - C_i)/2$ when compared to architecture A2 as demonstrated in Fig. 4.11.

Figure 4.10 Architecture A3 for Encoder stack.

The decoder follows a similar architecture with a minor variation in the load phase. Each decoder consists of an M-MHA block, an MHA block, and an FFN block. The load and compute latency of the two MHA blocks are approximately equal to the FFN block. Hence we load the combined weights of the two MHA blocks, overlapping with the load of the FFN block. The compute phases are executed sequentially.

Figure 4.11 Architecture A3 for Decoder stack.

LWi_m and Ci_m correspond to the load and compute time of the combined M-MHA and MHA blocks, respectively. LWi_f and Ci_f correspond to the load and compute time of the FFN block, respectively.

Note that when C_i is greater than LW_i, the stalls in the compute phase are eliminated. This occurs when the input sequence length, s is greater than 18, after which architectures A2 and A3 perform

similarly, ensuring no stalling in the compute phase. The Load time remains nearly constant with an increase in sequence length, as the data size of the weight matrices remains constant. However, the MM_i and other computations require higher latency and resources with increased input sequence length.

4.6 End-to-End control flow

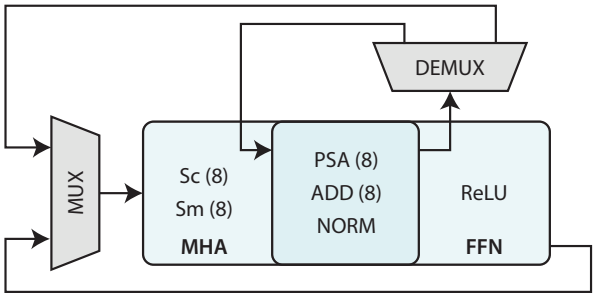

Figure 4.12 Top level controller design.

The main controller sequentially orchestrates the flow through multiplexed MHA and FFN blocks, realizing an end-to-end Transformer architecture (refer to Fig. 4.12).

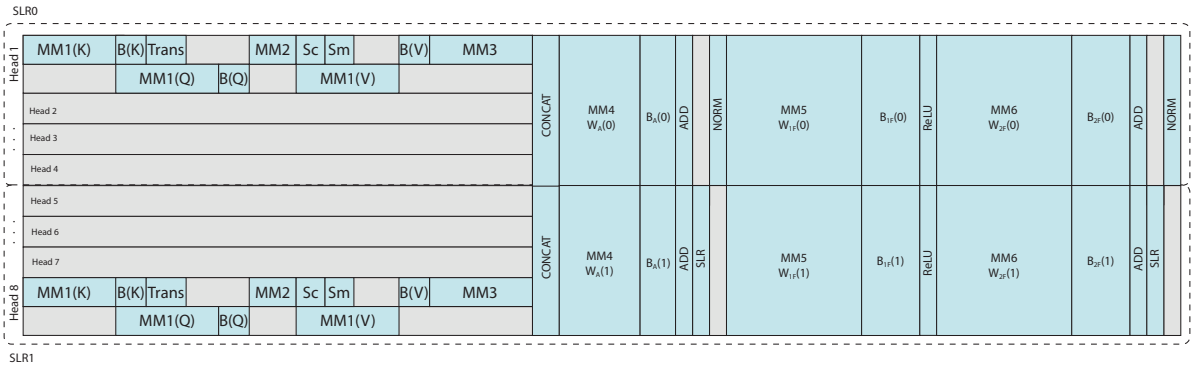

Figure 4.13 Block-wise scheduling of operations within an encoder

Fig. 4.13 illustrates the block-wise scheduling of operations in an encoder while mitigating inter-SLR communication. The MM_1 block is sequentially utilized within each attention head, while the eight attention heads are executed concurrently, computing 4 attention heads within each SLR. The computational flow begins with calculating the Key matrix (K). The $s \times 64$ adder performs the Bias operation, B(K), in parallel with the query matrix multiplication, MM_1(Q). The MM_2 and MM_1(V) operations are performed sequentially, with both of them utilizing the same PSA block. The scaling (Sc) and softmax (Sm) operations are launched in parallel to MM_1(V), as the combined latency of the operations ($t_{Sc} + t_{Sm}$) is less than that of MM_1(V) operation. Hence by scheduling the operations

in parallel, the latency of each attention head is reduced without expending additional resources. We perform bias B(V) followed by MM_3 using the same PSA structure.

The resultant matrices from the four attention-heads in each SLR are concatenated. The concatenated matrices, along with a partitioned weight matrix $W_A(0/1)$ are the inputs of MM_4. Similarly, we perform $B_A(0/1)$ using the $s \times 64$ adders followed by Add-Norm. Addition at Bias in the linear layers outside the attention-heads is done across the two SLRs utilizing the eight parallel adders. The residual operations in the FFN namely, MM_5, B_{1F}, $ReLU$, MM_6, and B_{2F} are performed in a similar fashion.

The Add-Norm block is executed as independent Add and Norm operations. The Add block splits the matrix addition of two $s \times 512$ matrices over both the SLRs. The resultant matrix followed by concatenation and normalization (Norm) is communicated to the global memory for further processing by the host. The control flow ensures the PSA blocks, which perform the major portion of computation run for the entire time frame except for minute stalls during additions, concatenations, and normalization. The eight $s \times 64$ adders primarily operate within the MM_i blocks along with ADD (Bias) and Add-Norm operation (refer Fig. 3.2 and Fig. 3.3).

Chapter 5

Experiments and Results

5.1 Experimental Results

The design is implemented on an Alveo U50 FPGA card with the latencies obtained at an operational frequency of 300MHz.

5.1.1 Output of the ASR system

In ESPnet, two versions of the Transformer model are provided - ”transformer_base” and ”transformer_large”. We use the ”transformer_base” model, which is a smaller version of the Transformer architecture. It is designed for faster training and lower computational cost without significantly compromising the accuracy of the speech recognition task. As discussed in Chapter 3.1, the E2E flow consists of data preparation of the audio file, followed by feature extraction, where the feature vectors are generated, followed by decoding using the Transformer architecture to generate a textual output as shown in Fig. 5.1.

```
stage 0: Data preparation
stage 1: Feature Generation
steps/make_fbank_pitch.sh --cmd run.pl --nj 1 --write_utt2num_frames true decode/6377-34630-0045/data decode/6377-34630-0045/log decode/6377-3
steps/make_fbank_pitch.sh: moving decode/6377-34630-0045/data/feats.scp to decode/6377-34630-0045/data/.backup
utils/validate_data_dir.sh: WARNING: you have only one speaker.
   Search for the word 'bold' in http://kaldi-asr.org/doc/data_prep.html
   for more information.
utils/validate_data_dir.sh: Successfully validated data-directory decode/6377-34630-0045/data
steps/make_fbank_pitch.sh: [info]: no segments file exists: assuming wav.scp indexed by utterance.
steps/make_fbank_pitch.sh: Succeeded creating filterbank and pitch features for data
/home/yamini/espnet/egs/librispeech/asr1/../../../utils/dump.sh --cmd run.pl --nj 1 --do_delta
false decode/6377-34630-0045/data/feats.scp decode/download/librispeech.transformer.v1/data/train_960/cmvn.ark decode/6377-34630-0045/log deco
stage 2: Json Data Preparation
stage 3: Decoding
6377-34630-0045.wav
/home/yamini/espnet/egs/librispeech/asr1
run.pl

Recognized text: _THE_PUBLIC_FOLLOW_WITH_GUSTO_THE_SCENT_OF_ANYTHING_CONTRABAND_TO_BE_SUSPECTED_IS_A_RECOMMENDATION_THE_PEOPLE_ADOPT_BY_INSTIN

Finished
```

Figure 5.1 Textual output from raw audio

The predicted transcriptions are evaluated against the ground truth transcriptions using metrics such as Word Error Rate (WER). We measure a WER of $\sim 9.5\%$ for our model.

5.1.2 Load-compute analysis

Here, we study the influence of input size on the load and compute cycle.

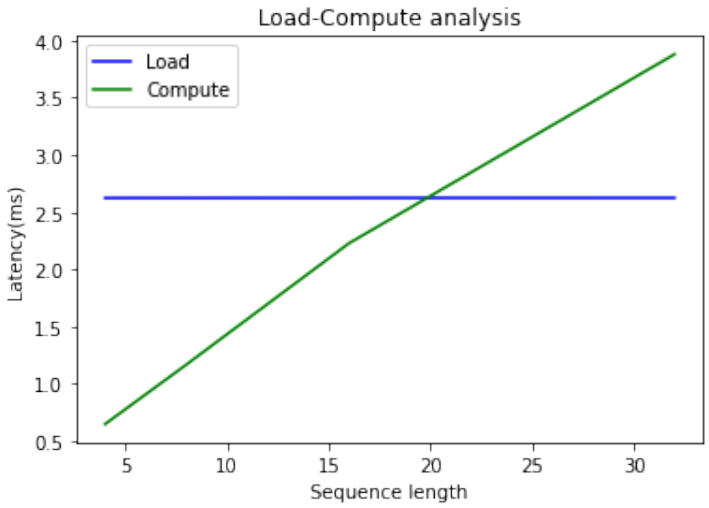

Figure 5.2 Comparison of load time and compute time of one MHA + FFN block

From Fig. 5.2, we observe that the compute time exceeds the Load time for a sequence length $s > 18$. Upon analyzing the influence of sequence length on the load and compute time, we observe that the load time remains nearly constant with an increase in sequence length, as the data of the weight matrices remains the same. However, the latency of the compute phase significantly varies with the input size.

5.1.3 Architecture analysis and comparison

Table 5.1 compares the performance of the three architectures on the latency metric on input sequences of varying lengths. The architecture A3 gives a 1.46x to 1.94x speedup compared to A1 due to the overlap between computation and communication. The performance improvement is more significant for lower sequence length inputs. A3 performs better than A2 when the latency of the Load phase is greater than the Compute phase, which occurs when the input audio is greater than $\sim$8 seconds.

Table 5.2 depicts the maximum resource utilization when the design is synthesized for input sequence length 32. The available resources on the device are approximately equally distributed between the two SLR regions. It can be observed that the DSP utilization is relatively low. We cannot improve this as this exerts the available FFs and LUTs, making the design unsynthesizable.

Table 5.1 Architecture-wise latency comparison for sequence lengths: 4, 8, 16, and 32

Sequence length	*Architecture*	*Latency (ms)*	*Improvement*
4	A1	65.87	1×
	A2	53.45	1.23×
	A3	33.92	1.94×
8	A1	75.57	1×
	A2	54.5	1.38×
	A3	39.9	1.89×
16	A1	98.14	1×
	A2	56.27	1.74×
	A3	52.59	1.86×
32	A1	122.8	1×
	A2	84.15	1.46×
	A3	84.15	1.46×

Table 5.2 Resource Utilization for sequence length 32

Resources	*BRAM_18K*	*DSP*	*FF*	*LUT*
Resources utilized	1202	1348	1191892	765828
Available Resources	2688	5952	1743360	871680

5.1.4 Discussion

Fig. 4.13 suggests the conservative block-level parallelism within the attention blocks. With the available DSPs, a $\sim$2.5× improvement can be achieved with the parallel usage of eight MM_1 blocks per SLR, enabling the concurrent calculation of Query and Key projections. However, the architecture is limited by the LUTs since the processing elements within the systolic array structure are LUT-intensive. Within the resource constraints, we have experimented on architectures with four, two, and one attention head in parallel, with two, four, and eight concurrent MM_1 blocks in each head, respectively, as shown in Table 5.3.

Table 5.3 Design space exploration

Number of parallel heads	**Concurrent PSA blocks per head**	**Latency (ms)**
8	1	84.15
4	2	85.72
2	4	87.43
1	8	92.03

We have also experimented with various dimensions of the PSA block with different unroll factors. Finally, we allocate maximum resources to matrix multiplication, the slowest block, while achieving

parallelism within the attention heads. Thus, the proposed design presents an optimal latency by exploiting the available resources. We observe that the FFN block containing larger matrix multiplication operations consumes approximately double the latency compared to the MHA block for a given sequence length.

5.1.5 Performance Comparison with CPU and GPU

Table 5.5 evaluates the latencies and performance improvement of the proposed accelerator compared to the latency of a server-based Intel Xeon E5-2640 CPU @ 2.5GHz with 24 cores and 64GB RAM. The software implementation is based on wav2vec, which is built on PyTorch. Table 5.5 also compares the latencies and performance improvement with an NVIDIA GeForce RTX 3080 Ti GPU @ 1.37GHz with 12GB RAM built on Pytorch and CUDA 10.1.

Table 5.4 Latencies for different sequence length inputs compared to a CPU

Input sequence length: i	**CPU latency in seconds**	**Latency improvement (CPU)**
4	0.4	4.75×
8	1.1	13.1×
16	3.1	36.8×
20	3.4	40.5×
24	3.8	45.2×
32	4.5	53.5×

Table 5.5 Latencies for different sequence length inputs compared to a GPU

Input sequence length: i	**GPU latency in seconds**	**Latency improvement (GPU)**
4	0.34	4.01×
8	0.46	5.4×
16	0.55	6.3×
20	0.79	9.39×
24	1.03	12.1×
32	1.32	15.5×

For six test inputs of sequence lengths 4, 8, 16, 20, 24, and 32, the average performance improvement noted compared to the CPU is $32\times$.

Similarly, for six test inputs of various sequence lengths, the average performance improvement noted compared to the GPU is $8.8\times$, while the accuracy remains the same.

The audio files of the LibriSpeech dataset range from 1 to 15 seconds. We have experimented the same with audio files up to 13 seconds. The hardware design, once implemented, handles inputs of fixed sequence length s. For a given input sequence of length i, where $i < s$, the input is padded up to s. Hence, with test input samples of varying sequence lengths, and a maximum sequence length of 32, we achieve performance improvement in the range of $4.75\times$ to $53.5\times$ compared to the CPU. The performance improvement is between $4.01\times$ to $15.5\times$ compared to the GPU.

5.1.6 Other results

Fig. 5.3 shows the platform diagram of the Alveo U-50 accelerator loading a set of weights for one encoder/decoder from the host onto the device through the PCIe interface. The kernel on each SLR loads data from an HBM channel. Each kernel loads weights from 2 HBM channels in parallel for smaller sequence lengths to hide the communication latency. When a kernel reads data from multiple HBM channels, it issues separate memory access requests for each channel to the memory interface. The memory interface will then manage the memory access to each HBM channel separately and transfer the data to the kernel as requested.

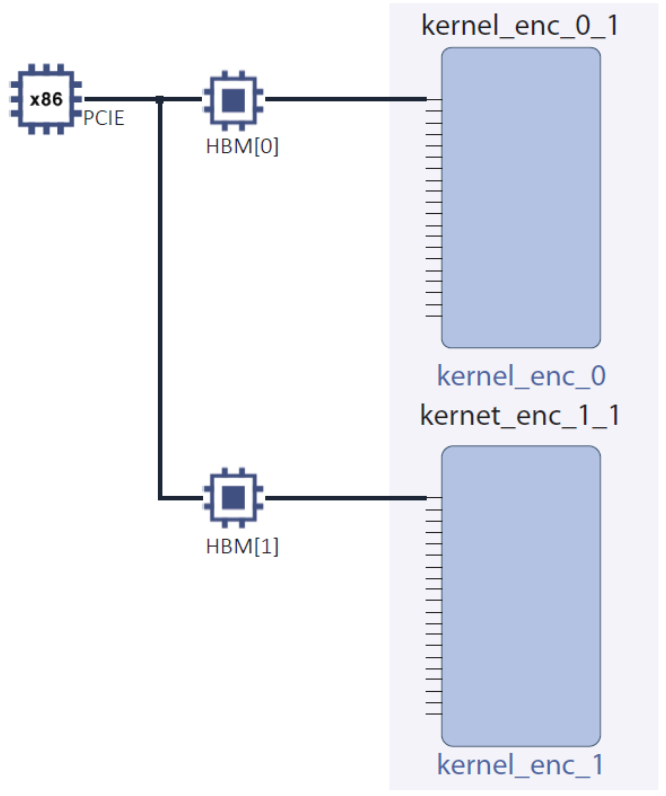

Figure 5.3 Platform diagram of Alveo U-50

The E2E ASR system demonstrates an overall latency of 120.45ms for a given input sequence of length 32. The combined latency of pre-processing and data preparation of the audio samples on the host is 36.3 ms. The overall achievable throughput of the E2E system is 11.88 sequences/second. The energy efficiency of the FPGA is 1.38 GFLOPs/J which is considerably high compared to the GPU's performance per joule, which is $\sim$0.055 GFLOPs/J.

5.1.7 Performance Comparison with other works

We cannot provide an objective comparison with different models of varying dimensions, hyper-parameters, and input sequence lengths.

Hence, we measure the performance in terms of GFLOPs/s (Giga floating-point operations per second) as a metric for a fair perspective. We compare with other reference works, comparing a CPU [34], GPU [29], and FPGA [29] implementation, respectively, as given in Table 5.6.

Table 5.6 Performance comparison with reference works

Parameters	**[34] CPU**	**[29] GPU**	**[29] FPGA**	**Our work FPGA**
GFLOPs	1.1	1.1	0.114	4.0
Latency (s)	2.1	0.147	0.00785	0.08415
GFLOPs/Latency	0.52	7.48	14.47	47.23
Improvement	1$\times$	14.38$\times$	27.82$\times$	90.8$\times$

The model deployed in [29] consists of 2 encoders and one decoder layer; the hidden size is 400; the feed-forward size is 200, with four attention heads. The latencies are reported on an 8$\times$NVIDIA Quadro RTX 6000 GPU and an Alevo U200 FPGA. From Table 5.6, we can observe a performance improvement (GFLOPs/s) of 90.8$\times$ compared to an ARM CPU. We measure a 6.31$\times$ and 3.26$\times$ improvement compared to a GPU and FPGA [29] implementation, respectively.

Chapter 6

Relevant Work and Conclusion

6.1 Related Work

Due to widespread applications of deep neural networks in solving many vision problems both on the edge and cloud, there is a vast body of literature on accelerating CNNs using GPUs; and hardware accelerators based on ASICs [5, 16] and FPGAs [38, 39]. Despite the Transformers widely replacing the RNN and CNN-based models in NLP and Vision applications, very few works on accelerating Transformers have been proposed. Transformers for NLP applications, where the inputs are discrete tokens, have been widely explored compared to ASRs, where the inputs are continuous frame-level operations. Wang et al. [35] proposed an ASIC accelerator for vision transformers involving only encoders. Li et al. [18] proposed an energy-efficient accelerator for an NLP Transformer using an enhanced block circulant matrix approach for the compression of weights. They evaluated the accelerator on two small models, one involving an encoder-decoder pair and another consisting of 12 encoders with no decoders. A Transformer based accelerator for Neural Machine Translation tasks was recently proposed by Lu et al. [22]. The thesis explains the independent block-wise implementation of the quantized Multi-Head Attention and Feed Forward Network blocks with an efficient design for non-linear operations like softmax and layer-norm. The design achieves significant latency improvement for a given sequence length. Li et al. [20] propose a Vision Transformer accelerator with a mixture of fixed-point and power-of-two quantization. The DSP and LUT cost is estimated to find the best ratio of fixed-point to PoT quantized weights while maximizing the throughput. Peng et al. [28] propose a pruning algorithm by comparing various sparse matrix formats, individually designing FPGA accelerators for encoder and decoder layers. Qi et al. [29] propose an NLP transformer accelerator by model pruning and corresponding hardware optimizations for sparse matrix storage and processing. As discussed in Subsection 5.1.7, the transformers used in different works differ in terms of the number of attention heads, the number of encoders/decoders, quantization, input sequence length, embedding size, etc., which makes it challenging to provide an objective comparison with our accelerator processing an E2E ASR transformer with 12 encoders and 6 decoders. However, as mentioned earlier, our work achieves a comparable latency with Li et al. [18], who conducted experiments on relatively smaller models.

6.2 Conclusion and future work

In this thesis, we propose an accelerator for a 32-bit floating point single precision Transformer model. We have explored FPGA-based hardware accelerators for end-to-end automatic speech recognition, and it is the first work in this space to the best of our knowledge.

All the matrix multiplications are routed on the systolic-array engines while enabling maximum reuse of resources between various matrix multiplications and also between the MHA and FFN blocks. The architecture is flexible as it can accommodate Transformer architectures of various dimensions. We can also perform device-specific customization by varying the PSA dimensions according to the available resources. We optimize the accelerator architecture with a carefully designed inter-SLR resource scheduling flow to minimize the stalling. We have compared three end-to-end architectures with short to medium-length sequences for an E2E ASR system.

The accelerator significantly outperforms a CPU-based server and a GPU with an average of $32\times$ and $8.8\times$ improvement in latency, with the MHA and FFN blocks independently sustaining $\sim$50 GFLOPs/s each. The accelerator shows a performance improvement in terms of latency when compared to other relevant works. The implementation has an accuracy or word error rate (WER) of $\sim$9.5, which is comparable to standard speech recognition systems. The architecture achieves an optimal trade-off between resource utilization and latency, suitable for real-time scenarios. In terms of future work, we will explore fixed precision end-to-end ASR models with no loss of accuracy. Fixed precision models offer lower resource utilization, addressing our primary constraint of LUT resources. This will enable the development of accelerators with lower latency, which is also suitable for edge deployment.

Related Publications

- D Shaarada Yamini, Mirishkar Ganesh S, Vuppala Anil Kumar, and Purini Suresh. ”Hardware Accelerator for Transformer based End-to-End Automatic Speech Recognition System.” In 2023 30th Reconfigurable Architectures Workshop (RAW).

Bibliography

[1] https://www.xilinx.com/products/boards-and-kits/alveo/u50.html.

[2] U. Banerjee, C. Juvekar, A. Wright, A. P. Chandrakasan, et al. An energy-efficient reconfigurable dtls cryptographic engine for end-to-end security in iot applications. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 42–44. IEEE, 2018.

[3] A. Bie, B. Venkitesh, J. Monteiro, M. A. Haidar, and M. Rezagholizadeh. Fully quantizing a simplified transformer for end-to-end speech recognition. 11 2019.

[4] A. J. R. D. Bie, B. Venkitesh, J. Monteiro, M. A. Haidar, and M. Rezagholizadeh. A simplified fully quantized transformer for end-to-end speech recognition. *arXiv: Computation and Language*, 2019.

[5] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *SIGARCH Comput. Archit. News*, 44(3):367–379, jun 2016.

[6] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, 34(2):171–210, 2002.

[7] L. Deng and D. Yu. Deep learning: Methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[8] L. Dong, S. Xu, and B. Xu. Speech-transformer: A no-recurrence sequence-to-sequence model for speech recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5884–5888, 2018.

[9] L. Dong, S. Xu, and B. Xu. Speech-transformer: A no-recurrence sequence-to-sequence model for speech recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5884–5888, 2018.

[10] L. Dong, S. Xu, and B. Xu. Speech-transformer: A no-recurrence sequence-to-sequence model for speech recognition. *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5884–5888, 2018.

[11] L. Gan, M. Yuan, J. Yang, W. Zhao, W. Luk, and G. Yang. High performance reconfigurable computing for numerical simulation and deep learning. *CCF Transactions on High Performance Computing*, pages 1–13, 2020.

[12] A. D. George and C. M. Wilson. Onboard processing with hybrid and reconfigurable computing on small satellites. *Proceedings of the IEEE*, 106(3):458–470, 2018.

[13] A. Gulati, J. Qin, C.-C. Chiu, N. Parmar, Y. Zhang, J. Yu, W. Han, S. Wang, Z. Zhang, Y. Wu, and R. Pang. Conformer: Convolution-augmented transformer for speech recognition. pages 5036–5040, 10 2020.

[14] R.-H. Hsu, J. Lee, T. Q. Quek, and J.-C. Chen. Reconfigurable security: Edge-computing-based framework for iot. *IEEE Network*, 32(5):92–99, 2018.

[15] M. Jiang, M. Jong, W. Lau, C. Chai, and N. Wu. Using automatic speech recognition technology to enhance efl learners' oral language complexity in a flipped classroom. *Australasian Journal of Educational Technology*, 37:110–131, 05 2021.

[16] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, jun 2017.

[17] M. Kim and Y. S. Shao. Hardware acceleration. *IEEE Micro*, 38(6):6–7, 2018.

[18] B. Li, S. Pandey, H. Fang, Y. Lyv, J. Li, J. Chen, M. Xie, L. Wan, H. Liu, and C. Ding. Ftrans: Energy-efficient acceleration of transformers using fpga. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '20, page 175–180, New York, NY, USA, 2020. Association for Computing Machinery.

[19] J. Li. Recent advances in end-to-end automatic speech recognition. *APSIPA Transactions on Signal and Information Processing*, April 2022.

[20] Z. Li, M. Sun, A. Lu, H. Ma, G. Yuan, Y. Xie, H. Tang, Y. Li, M. Leeser, Z. Wang, X. Lin, and Z. Fang. Auto-vit-acc: An fpga-aware automatic acceleration framework for vision transformer with mixed-scheme quantization, 2022.

[21] J. W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. Vissers. A low-latency library in fpga hardware for high-frequency trading (hft). In *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*, pages 9–16, 2012.

[22] S. Lu, M. Wang, S. Liang, J. Lin, and Z. Wang. Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer. In *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*, pages 84–89, 2020.

[23] R. Machupalli, M. Hossain, and M. Mandal. Review of asic accelerators for deep neural network. *Microprocessors and Microsystems*, 89:104441, 2022.

[24] S. A. Manavski. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In *2007 IEEE International Conference on Signal Processing and Communications*, pages 65–68, 2007.

[25] I. Martínez-Nicolás, T. E. Llorente, F. Martínez-Sánchez, and J. J. G. Meilán. Ten years of research on automatic voice and speech analysis of people with alzheimer's disease and mild cognitive impairment: A systematic review article. *Frontiers in Psychology*, 12, 2021.

[26] N. Moritz, T. Hori, and J. Le. Streaming automatic speech recognition with the transformer model. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6074–6078, 2020.

[27] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur. Librispeech: An asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210, 2015.

[28] H. Peng, S. Huang, T. Geng, A. Li, W. Jiang, H. Liu, S. Wang, and C. Ding. Accelerating transformer-based deep learning models on fpgas using column balanced block pruning. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, pages 142–148, 2021.

[29] P. Qi, Y. Song, H. Peng, S. Huang, Q. Zhuge, and E. H.-M. Sha. Accommodating transformer onto fpga: Coupling the balanced model compression and fpga-implementation optimization. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, GLSVLSI '21, page 163–168, New York, NY, USA, 2021. Association for Computing Machinery.

[30] J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu. Roformer: Enhanced transformer with rotary position embedding, 2022.

[31] M. Vardhana, N. Arunkumar, S. Lasrado, E. Abdulhay, and G. Ramirez-Gonzalez. Convolutional neural network for bio-medical image segmentation with hardware acceleration. *Cognitive Systems Research*, 50:10–14, 2018.

[32] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[33] D. Wang, K. Xu, J. Guo, and S. Ghiasi. Dsp-efficient hardware acceleration of convolutional neural network inference on fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4867–4880, 2020.

[34] H. Wang, Z. Wu, Z. Liu, H. Cai, L. Zhu, C. Gan, and S. Han. Hat: Hardware-aware transformers for efficient natural language processing. 05 2020.

[35] H.-Y. Wang and T.-S. Chang. Row-wise accelerator for vision transformer. In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 399–402, 2022.

[36] S. Watanabe, T. Hori, S. Karita, T. Hayashi, J. Nishitoba, Y. Unno, N. Yalta, J. Heymann, M. Wiesner, N. Chen, A. Renduchintala, and T. Ochiai. Espnet: End-to-end speech processing toolkit. *ArXiv*, abs/1804.00015, 2018.

[37] C.-F. Yeh, J. Mahadeokar, K. Kalgaonkar, Y. Wang, D. Le, M. Jain, K. Schubert, C. Fuegen, and M. Seltzer. Transformer-transducer: End-to-end speech recognition with self-attention, 10 2019.

[38] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He. Opu: An fpga-based overlay processor for convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(1):35–47, 2020.

[39] Y. Yu, T. Zhao, K. Wang, and L. He. Light-opu: An fpga-based overlay processor for lightweight convolutional neural networks. FPGA '20, page 122–132, New York, NY, USA, 2020. Association for Computing Machinery.

[40] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, page 161–170, New York, NY, USA, 2015. Association for Computing Machinery.