

Language Model Inference on FPGA with Integer-only Operations

by

Marat Bekmyrza

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2025

© Marat Bekmyrza 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Large Language Models (LLMs) are currently dominating the field of Artificial Intelligence (AI) applications, but their integration for edge computing purposes is rather limited due to computational complexity and power consumption. This thesis addresses this challenge by investigating the integer-only acceleration of transformer models on FPGAs, focusing on the BERT architecture. We demonstrate that by removing the floating-point operations from the inference pipeline, especially from non-linear functions like GELU, Softmax, and Layer Normalization, we can improve the performance without sacrificing accuracy.

Our pipelined, batched architecture processes multiple sequences in parallel and optimizes the FPGA resources. We achieve a 2.6x throughput improvement over a single-sequence inference and at least 10x speedup over the offloading to CPU approach. The results of the experiments show that our implementation has comparable accuracy to the floating-point models for the GLUE benchmark tasks with INT8 quantization. These findings reveal that integer-only transformer inference on FPGAs is a feasible way of implementing complex language models on resource-limited edge devices, with the potential for new privacy-conscious, low-latency AI applications.

Acknowledgements

I am grateful to my supervisors, Professor Nachiket Kapre and Professor Hiren Patel, for the opportunity to join their research groups and for their guidance throughout my graduate studies. I also want to thank the members of the Waterloo Configurable Architectures Group and the Computer Architecture and Embedded Systems Research team at the University of Waterloo. In particular, I want to acknowledge Omar Elayat, Emir Guevara, Jehanzeb Mirza, and Joel Deodhar, for the great discussions and feedback.

Table of Contents

Author's Declaration	ii
Abstract	iii
Acknowledgements	iv
List of Figures	viii
List of Tables	x
1 Introduction	1
1.1 Thesis Contributions	3
1.2 Thesis Outline	3
2 Background	4
2.1 Transformer Model	4
2.1.1 Architecture	6
2.1.2 Attention Mechanism	6
2.2 BERT Model	8
2.2.1 Architecture	8
2.2.2 Concept of Pre-training	8
2.3 Quantization	9

2.3.1	Quantization Methods	9
2.3.2	Quantization Strategies	11
2.4	I-BERT	11
2.4.1	GELU	13
2.4.2	Softmax	14
2.4.3	Layer Normalization	18
3	Related Work	22
3.1	Transformer Quantization	22
3.2	FPGA-based Transformer Accelerators	23
3.3	Nonlinear Functions	24
4	Implementation	25
4.1	Model Computation Flow	25
4.2	Linear Layer	28
4.2.1	Systolic Array	28
4.2.2	Requantization	29
4.3	Accelerator Architecture	30
4.3.1	Attention Head	30
4.3.2	Linear Layer and Layer Normalization	31
4.3.3	Linear Layer and GELU	32
4.3.4	Encoder	33
4.4	Architecture Optimizations	33
4.4.1	Slicing	33
4.4.2	One Systolic	34
4.4.3	Read Residuals	35
4.4.4	Block-wise Output	36
4.4.5	AXI Packing	36

4.5	Batch Processing	36
4.5.1	Pipelined Execution	37
4.5.2	Pipeline Buffers	37
4.6	FPGA Optimizations	39
4.6.1	Memory Optimizations	39
4.6.2	DSP Packing	42
5	Results	44
5.1	Training	44
5.2	Inference	47
5.2.1	Design Space Exploration	48
5.2.2	Runtime Comparisons	50
5.2.3	Resource Utilization	51
5.2.4	CPU Offloading	52
5.2.5	Power Consumption	53
5.3	Comparison with Prior Works	54
6	Conclusion	57
6.1	Conclusion	57
6.2	Limitations and Future Work	58
	References	59

List of Figures

2.1	Architecture of the Transformer model. Figure adopted from [33].	5
2.2	Symmetric quantization maps the input range $[-\max(x), \max(x)]$ to quantized range $[-127, 127]$ for 8-bit quantization. Red dots represent example mappings from the input range to the quantized range.	9
2.3	Asymmetric quantization maps the input range $[\min(x), \max(x)]$ to quantized range $[0, 255]$ for 8-bit quantization. Red dots represent example mappings from the input range to the quantized range.	10
2.4	I-BERT Quantization	12
2.5	Dataflow of the i-GELU function	15
2.6	Dataflow of the i-exp function	17
2.7	Dataflow of i-Softmax function	18
2.8	Dataflow of i-LayerNorm function	20
4.1	Computation flow of a BERT encoder layer.	27
4.2	Dataflow of the <code>attn_head</code> module	31
4.3	Dataflow of the <code>mm_ln</code> module	32
4.4	Dataflow of the <code>mm_gelu</code> module	32
4.5	Dataflow of the <code>encoder</code> module	33
4.6	Dataflow of the Slicing optimization	34
4.7	Dataflow of the One Systolic optimization	35
4.8	Pipelined batch processing	37

4.9	Encoder layer with residual and output buffers. Orange boxes represent buffers.	38
4.10	Execution Waveforms: (a) fully sequential design, (b) perfectly balanced pipelined design with batch processing, (c) batch processing with wasted cycles due to absence of residual and output buffers, (d) batch processing with residual buffers, but without output buffers, (e) batch processing with residual and output buffers	40
5.1	Validation accuracy of I-BERT-base model during training on MRPC and QNLI tasks	46

List of Tables

4.1	Dimensions of model input and parameters.	28
4.2	Sequential execution pattern showing each sequence (S0-S3) completing all modules 12 times before the next sequence begins. A, S, I, and L represent Attention Head, Self Attention Output, Intermediate, and Layer Output modules respectively.	38
4.3	Pipelined execution pattern showing concurrent processing of different sequences across modules. Each sequence (S0-S3) flows through the pipeline while allowing multiple modules to be active simultaneously.	38
5.1	Description of GLUE benchmark tasks	45
5.2	Performance comparison of different BERT base models on GLUE benchmark tasks (higher is better). I-BERT results are taken from [17], whereas I-BERT-f is the performance result of our finetuning.	46
5.3	Performance comparison of different BERT base models on GLUE benchmark tasks.	46
5.4	Hardware specifications of evaluated platforms	47
5.5	FPGA platform specifications	47
5.6	Dimensions of I-BERT [17] models in FP32 baseline	47
5.7	Impact of number of slices on runtime performance on PYNQ-Z1	48
5.8	Our FPGA inference designs with different frequencies, systolic array dimensions, and AXI packing.	49
5.9	Performance comparison of non-batched and batched designs at different operating frequencies on KV260.	50

5.10	Attention head execution cycles	50
5.11	Impact of quantization on inference latency (ms)	51
5.12	Comparison of inference latency and throughput of different devices	52
5.13	FPGA resource utilization for the single and four AXI designs on KV260.	52
5.14	Resource utilization of nonlinear functions on FPGA.	53
5.15	CPU Offload Operation Runtimes in FP32	53
5.16	Power consumption and latency comparison across different devices, batch size of 1.	54
5.17	Platform and model specifications of different FPGA implementations	55
5.18	Resource usage and performance of different designs: Lat. is Latency (lower is better), Thr. is Throughput (higher is better), Thr./DSP is Throughput per DSP (higher is better). Notes: 1 – decoder based models report tokens per second.	55
5.19	Resource utilization comparison of different nonlinear function implementa- tions	56

Chapter 1

Introduction

Large Language Models (LLMs) have revolutionized the Natural Language Processing (NLP) and Artificial Intelligence (AI) applications across numerous domains. While significant research efforts have focused on training increasingly large models, attention is now shifting toward efficient inference. This thesis addresses the challenge of accelerating transformer model inference on Field-Programmable Gate Arrays (FPGAs) through integer-only computation.

Transformer-based language models have achieved remarkable performance in various natural language processing tasks [33, 9]. However, their computational and memory demands present challenges for practical deployment. While quantization techniques have successfully reduced the precision of linear operations (matrix multiplications) to 8-bit or 4-bit integers, most implementations still rely on floating-point arithmetic for nonlinear functions like GELU, Softmax, and Layer Normalization. This mixed-precision approach limits the full the potential of quantization benefits, as floating-point operations consume significantly more power than their integer counterparts - up to 30x for addition and 18.5x for multiplication operations compared to FP32 [7].

While power efficiency is a critical concern even in data center environments, the need for efficient inference becomes even more pressing in edge computing scenarios. Language models are increasingly being deployed on edge devices, since it enables operation with limited network connectivity reduces expenses by eliminating constant cloud service fees, and enhances privacy since data remains on the device. FPGAs are especially suitable for the acceleration of edge AI because they are reconfigurable, energy-efficient, and can implement custom data paths that are optimized for a particular workload. In contrast to ASICs, FPGAs can be reprogrammed after fabrication, enabling rapid updates as AI

workloads evolve, without costly hardware redesign.

For inference with quantized language models, nonlinear functions present a unique challenge. While linear operations (matrix multiplications, additions) can be straightforwardly implemented with integer arithmetic, nonlinear functions like GELU, Softmax and Layer Normalization traditionally require dequantization to floating-point. Recent work has demonstrated that replacing floating-point operations with integer-only polynomial approximations can significantly improve inference speed. I-BERT [17] showed speedups of 2.4x-4.0x on GPUs without accuracy loss through end-to-end integer computation. However, the application of integer-only nonlinear functions for FPGA acceleration remains largely unexplored. Building on the approach introduced in I-BERT for GPUs [17], we demonstrate that integer-only nonlinear function implementations are not only feasible but highly beneficial for FPGA acceleration, offering advantages in inference speed with minimal additional hardware resources.

Beyond arithmetic optimizations, improving system-level throughput is also crucial. We explore batching techniques that enable parallel processing of multiple input sequences, further improving throughput for applications where multiple inference requests must be processed simultaneously.

1.1 Thesis Contributions

This work presents the following contributions:

- Implementation and evaluation of integer-only nonlinear functions (GELU, Softmax, Layer Normalization) on FPGAs that can be integrated into various transformer architectures. Our implementation shows the benefits of executing nonlinear functions on FPGAs, achieving more than 10x speedup compared to offloading them to CPUs with only 10% of additional hardware resources.
- A novel batching approach for FPGA-based transformer inference that enables parallel processing of multiple input sequences. Our pipelined architecture with four channels achieves a 2.6x throughput increase across different operating frequencies over a single sequence inference.
- Comprehensive evaluation of the performance, resource utilization, and accuracy tradeoffs of integer-only inference on GPUs, Apple chips, and FPGA boards and design space exploration demonstrating the scalability of the proposed approach across different FPGAs and frequency targets.

1.2 Thesis Outline

The remainder of this text is organized as follows:

- Chapter 2 presents background on transformer models, quantization techniques, and the I-BERT architecture that serves as the foundation for our work.
- Chapter 3 reviews prior works on inference acceleration of transformer models on the FPGAs, and on techniques for implementing nonlinear functions.
- Chapter 4 describes the implementation of integer-only nonlinear functions using polynomial approximations, architecture design, batch processing, and optimizations made to reduce resource utilization and latency.
- Chapter 5 presents experimental results, including performance comparisons across different hardware platforms, and a resource utilization analysis.
- Chapter 6 summarizes the contributions and findings of the thesis, discusses limitations, and outlines opportunities for future work.

Chapter 2

Background

This chapter introduces background concepts essential to our work. We begin with the transformer architecture and its attention mechanism, followed by BERT and the concept of pre-training. We then explore quantization methods for neural networks. Finally, we discuss I-BERT’s integer-only inference strategy [17].

2.1 Transformer Model

The Transformer model, which was introduced in the seminal paper “Attention Is All You Need” by Vaswani et al. [33], outperformed previous state-of-the-art models in natural language processing (NLP) tasks and laid the foundation for the emergence of modern pre-trained LLMs (e.g., BERT, GPT family, Llama, etc.).

Prior to this work, NLP tasks were primarily addressed by networks based on recurrence, such as Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM), and Gated Recurrent Units (GRUs). The main limitation of recurrence-based networks is that the input data has to be processed sequentially, one time step at a time. The current hidden state of a recurrence-based network is computed as a function of the previous hidden state and the current input. This sequential processing constrains the parallelization opportunities of the model and makes it difficult to learn long-term dependencies due to the vanishing gradient problem. The lack of parallelization makes the training process of recurrent networks very slow.

The Transformer model, based on an encoder-decoder architecture, was proposed to replace recurrence-based networks through the introduction of an attention mechanism

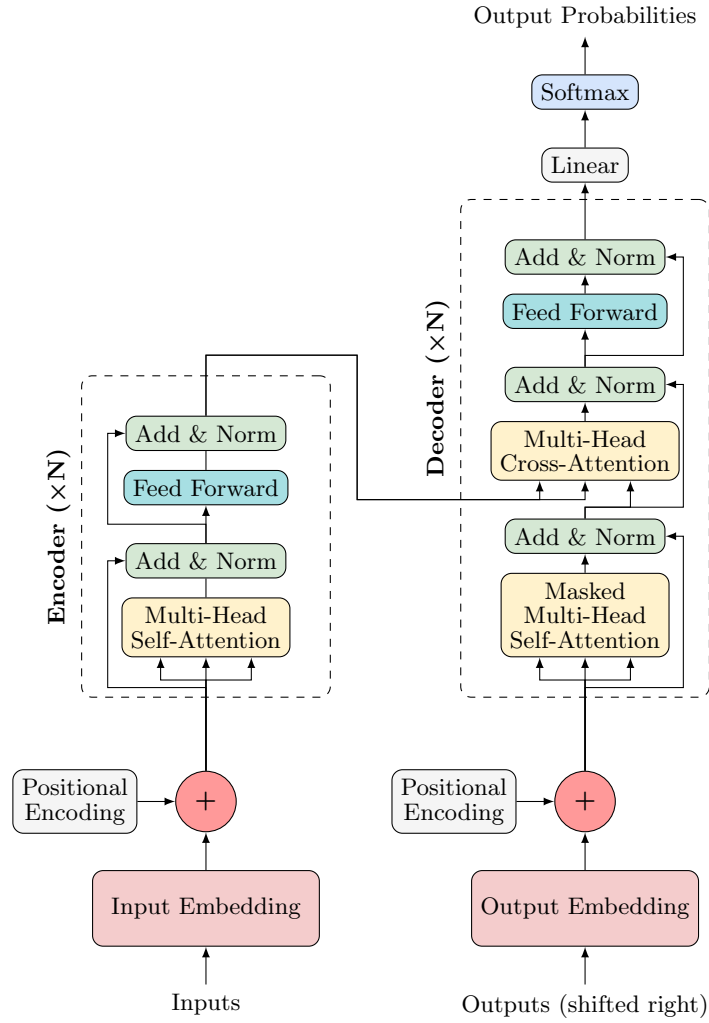


Figure 2.1: Architecture of the Transformer model. Figure adopted from [33].

that could process entire sequences at once. This parallelization property improves the training time of the model and also helps it learn long-range dependencies better than previous architectures.

2.1.1 Architecture

First, the input sequence, which can be text or any sequential data, is tokenized. Tokenization is the process of mapping each element of a sequence (e.g., a word, syllable, or punctuation mark) to a unique integer using a predefined dictionary.

Then, each token is mapped into a vector by using its integer value as an index to a lookup table, called an embedding matrix. To preserve the information about the relative order of the sequence elements, the authors in [33] proposed to compute positional encodings for each token based on its positional index in the sequence. The input embeddings and positional encodings are summed to produce an input matrix, which is then passed to the Encoder.

The Encoder is a stack of N identical layers, where each layer consists of two sub-layers: a multi-head self-attention module followed by a fully connected feed-forward network. Residual connections are employed around each of the two sub-layers, combined with layer normalization. The output of the final Encoder layer is connected to the multi-head cross-attention module of all Decoder layers. This multi-head cross-attention module of a Decoder layer is placed between the masked multi-head attention module and the fully connected feed-forward network. The masked multi-head attention module prevents the decoder from attending to future tokens during training. The output of the last Decoder layer is passed to a linear layer and a softmax function, which produces the final output of the model.

2.1.2 Attention Mechanism

Attention is a mechanism that is designed to learn contextual information from dependencies between tokens of a sequence. For example, in the sentence “The Windows computer was stuck on a blue screen because it was updating”, the word “it” refers back to “computer”, demonstrating how attention helps the model understand these contextual relationships between tokens.

Attention is a function of a query, a key, and a value. The mechanism can be considered as a search process that matches the query with the key and returns the value.

Scaled Dot-Product Attention

Scaled Dot-Product Attention has three inputs: a query Q , a key K , and a value V .

It computes the dot product of the query and key vectors, scales the result by a constant, applies a softmax function to the result, and then multiplies the result by the value vectors. The dot product can be seen as a measure of similarity between the query and key vectors, and the softmax function is used to normalize the result. The output of the softmax function is a vector of “probabilities” that can be used to compute the weighted sum of the value V vectors to generate the output.

In practice, the query, key, and value inputs are represented as matrices, where each row corresponds to a separate token in the sequence. The attention mechanism computes the dot product between the query vector of a given token and the key vectors of all tokens, capturing pairwise similarity scores. Optional masking is applied at the softmax input by adding “infinity” to the positions that should not be attended to. Thus, the softmax function will output zero probability for those positions.

Multi-Head Attention

In the multi-head attention module, the input I is supplied to three linear layers, which produce the query Q , key K , and value V matrices. Each linear layer has a different set of weights and biases, W_Q , W_K , W_V , b_Q , b_K , b_V . The authors in [33] propose to replicate these three linear layers and a scaled dot-product attention h times to construct h attention heads, and then concatenate the results.

Splitting the attention mechanism into multiple heads enables the model to “attend” to information from different representation subspaces and learn different aspects of the input sequence at the same time [33].

In the **multi-head self-attention module**, the query, key, and value matrices are derived from the same input matrix I .

In the **multi-head cross-attention module**, the query matrix Q is derived from the output of the Decoder layer, and the key and value matrices K and V are derived from the output of the Encoder layers.

2.2 BERT Model

Bidirectional Encoder Representations from Transformers (BERT), which was proposed by Devlin et al. in 2018 [9], was among the first pre-trained transformer-based models to surpass existing methods on several major NLP benchmarks of that time.

2.2.1 Architecture

BERT is an encoder-only model that has an identical architecture to the Encoder part of the Transformer model in Fig. 2.1. The authors in [9] designed BERT to have the same model size as OpenAI GPT for comparison purposes. While GPT is a decoder-only model, it has the same architecture as the Decoder part of the Transformer model, but without the multi-head cross-attention module since it does not have an Encoder part. The difference between BERT and GPT is that BERT uses bidirectional self-attention, which is the same as the multi-head self-attention module in the Transformer model, while GPT uses masked self-attention where every token can only attend to tokens to its left. The masked self-attention is needed when the goal is to generate a new sequence of tokens.

Another subtle difference between BERT and the original Transformer model is that BERT uses a GELU activation function instead of ReLU in the feed-forward network.

2.2.2 Concept of Pre-training

The main idea of having a pre-trained model is to learn a general language representation that can be later fine-tuned for a downstream task. BERT was the first model [9] that enabled fine-tuning for a specific task with just one extra output layer.

The model achieves this through an innovative “masked language modeling” (MLM) pre-training objective, where random words in the input are masked, and the model must predict missing words in a sentence by looking at the surrounding context in both the left and right directions. BERT also introduces a “next sentence prediction” task during pre-training, enabling the model to understand whether two sentences naturally follow each other in the text.

Pre-trained on a massive corpus of unlabeled text including the BooksCorpus (800M words) and Wikipedia (2,500M words), BERT learns deep contextual representations that can be fine-tuned for various downstream tasks. The authors achieved state-of-the-art results on GLUE benchmark tasks, including question answering, named entity recognition, and natural language inference.

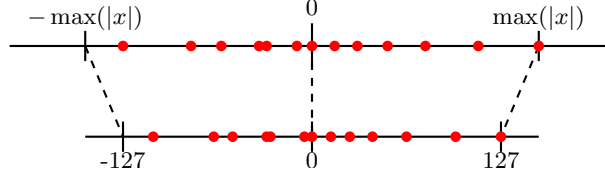


Figure 2.2: Symmetric quantization maps the input range $[-\max(|x|), \max(|x|)]$ to quantized range $[-127, 127]$ for 8-bit quantization. Red dots represent example mappings from the input range to the quantized range.

2.3 Quantization

Language models are trained using floating-point arithmetic for two main reasons: to make loss functions smooth and differentiable, and to ensure precise gradient calculations for accurate parameter updates during backpropagation. In order to do inference of a network, quantization is performed to reduce the number of bits used to represent the model weights and activations. This saves memory and increases the inference speed.

2.3.1 Quantization Methods

In this section, we focus on the quantization of real-valued (floating-point) tensors to integer values. There are two commonly used linear quantization methods based on the range of the input tensor: symmetric and asymmetric.

Symmetric Quantization

Symmetric Quantization uses a symmetric range around zero, simplifying computations but potentially wasting representation capacity.

As shown in Fig. 2.2, the maximum of the absolute value of the input tensor determines the input range that maps to the quantized range:

$$q = \text{Quantize}(x, S, M) := \text{Clamp}\left(\text{Round}\left(\frac{x}{S}\right), -M, M\right) \quad (2.1)$$

$$\text{Clamp}(x, a, b) := \min(\max(x, a), b) \quad (2.2)$$

where x is an input, q is the quantized output, S is the quantization scaling-factor, $\text{Round}(\cdot)$ denotes rounding to the nearest even integer, and M is the largest quantized value when

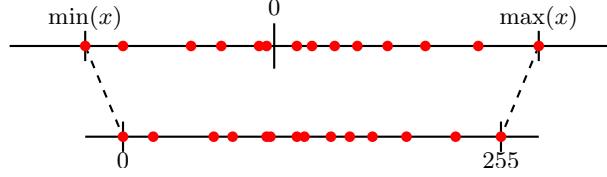


Figure 2.3: Asymmetric quantization maps the input range $[\min(x), \max(x)]$ to quantized range $[0, 255]$ for 8-bit quantization. Red dots represent example mappings from the input range to the quantized range.

quantizing to b bits:

$$M = 2^{b-1} - 1 \quad (2.3)$$

e.g., for 8-bit quantization, $M = 127$. The scaling factor is determined as follows:

$$S = \frac{\max(|x|)}{M} \quad (2.4)$$

Asymmetric Quantization

Asymmetric Quantization adapts to the actual value distribution, potentially providing better accuracy at the cost of more complex computations.

As shown in Fig. 2.3, in contrast to the symmetric quantization, here the actual minimum and maximum values of the input tensor determine the input range that maps to the quantized range:

$$q = \text{Quantize}(x, z, S, M) := \text{Clamp}\left(\text{Round}\left(\frac{x}{S}\right) - z, 0, M\right) \quad (2.5)$$

where z is the zero point, which is the quantization offset that maps the minimum value of the input tensor to 0 in the quantized space:

$$z = \text{Round}\left(\frac{\min(x)}{S}\right) \quad (2.6)$$

M is defined as:

$$M = 2^b - 1 \quad (2.7)$$

The scaling factor is determined as follows:

$$S = \frac{\max(x) - \min(x)}{M} \quad (2.8)$$

2.3.2 Quantization Strategies

There are two primary strategies for quantizing neural networks related to the training process:

- **Post-Training Quantization (PTQ)** lowers the precision of a pre-trained model without retraining. It requires minimal fine-tuning or calibration using a small dataset. While simpler to implement, PTQ may lead to accuracy degradation [40]. The main reason for accuracy degradation is usually due to outliers in the activations [38, 8]. However, there are techniques to improve the accuracy of PTQ by applying strategies to tackle the outliers in activation quantization [38].
- **Quantization-Aware Training (QAT)** employs quantization during the training process, allowing the model to learn to compensate for quantization errors. As demonstrated by [40], QAT typically achieves better accuracy compared to PTQ but requires full model retraining.

From the equations above, we see that the scaling factor is input-dependent. For weights and biases, this is set once training is complete. For activations, the minimum and maximum values can be calculated “online” during inference or “offline” using a calibration dataset.

- **Online Dynamic Quantization:** The quantization parameters are dynamic and change during inference. Scaling factors are calculated during inference for each tensor. While this prevents clipping issues, it introduces additional computational overhead.
- **Offline Static Quantization:** The quantization parameters are fixed and do not change during inference. Scaling factors are pre-computed using calibration data. This approach is more efficient but may suffer from clipping if runtime values exceed the calibrated range.

2.4 I-BERT

I-BERT is a model based on the pre-trained RoBERTa [24] that performs an end-to-end integer-only inference [17]. A key idea is that all the nonlinear functions (GELU, Softmax,

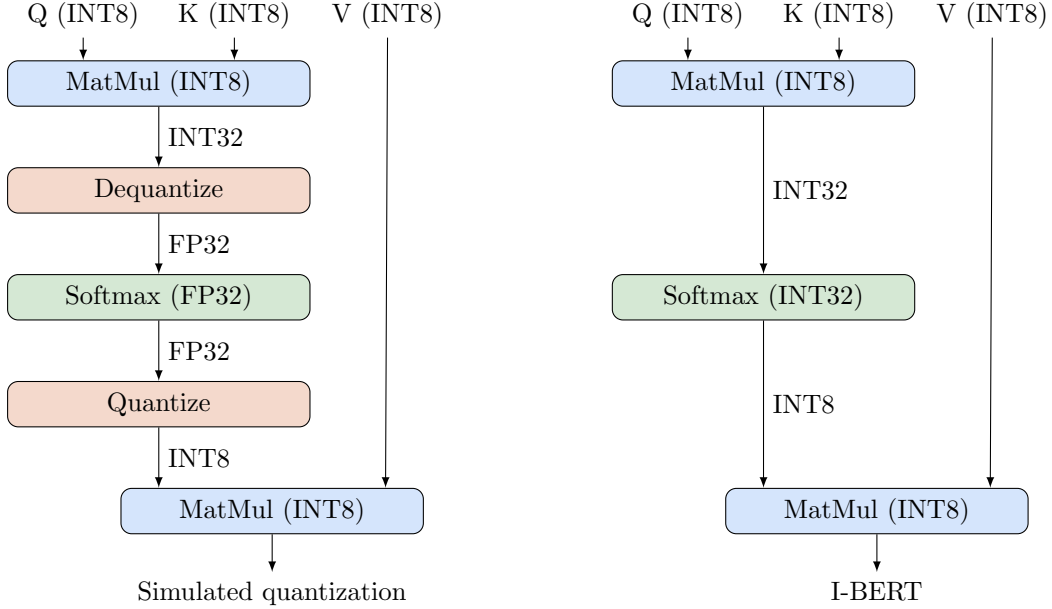


Figure 2.4: I-BERT Quantization

and Layer Normalization) are replaced by their integer-only polynomial approximations. The INT8 I-BERT model leads to similar accuracy as the FP32 baseline model with 2.4x-4.0x speedup in inference time on a T4 GPU. This indicates that it is possible to implement complex neural networks with all integer arithmetic without compromising the model performance. We use this strategy to implement integer-only inference on FPGAs.

Many hardware platforms, including FPGAs and edge devices, excel at integer arithmetic but perform floating-point operations inefficiently. While traditional quantized neural networks convert weights and activations to integers, they still retain floating-point operations in critical components, particularly in nonlinear functions. These floating-point operations create computational bottlenecks, requiring offloading to CPUs or dedicated hardware resources, limiting the efficiency gains from quantization. As shown on the left side of Fig. 2.4, the nonlinear operations are kept at FP32 precision for simulated quantization approach, where the activations are dequantized to FP32 precision before the nonlinear operations.

The uniform use of integer arithmetic enables more efficient memory access patterns allowing activations to stay fully on-chip. However, performing all operations with integer-only arithmetic presents unique challenges, particularly for nonlinear functions. While linear operations like matrix multiplication maintain their properties under quantization

$\text{MatMul}(qS) = S \cdot \text{MatMul}(q)$, nonlinear functions like GELU, Softmax, and LayerNorm do not follow this pattern [15, 39]. For instance, $\text{GELU}(qS) \neq S \cdot \text{GELU}(q)$. Therefore, nonlinear approximations should be carefully analyzed to ensure the correct transformation of the inputs and scaling factors.

I-BERT matrix multiplications use INT8 precision with INT32 accumulation, while nonlinear operations maintain INT32 precision to preserve accuracy. After nonlinear operations, requantization is performed to bring the precision back to INT8 for subsequent computations [39]. Static quantization is used, where scaling factors are precomputed during training and fixed during inference.

2.4.1 GELU

Gaussian Error Linear Unit (GELU) is a nonlinear activation function used in the BERT model [33]:

$$\text{GELU}(x) = x \cdot \frac{1}{2} \left[1 + \text{erf} \left(\frac{x}{\sqrt{2}} \right) \right] \quad (2.9)$$

$$\text{where } \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2) dt \quad (2.10)$$

where $\text{erf}(x)$ is the error function. I-BERT [17] approximates $\text{erf}(x)$ using a second-order polynomial $L(x)$:

$$L(x) = \text{sgn}(x) [a(\text{clip}(|x|, \max = -b) + b)^2 + 1] \quad (2.11)$$

where $a = -0.2888$ and $b = -1.769$, and $\text{sgn}(x)$ denotes the sign function. Note that $\text{clip}(|x|, \max = -b)$ limits the input to the range $[-1.769, 1.769]$, which is justified since $\text{erf}(x)$ approaches 1 and -1 for values of x outside this range. Replacing $\text{erf}(x)$ with $L(x)$ in Equation 2.9, the integer-only i-GELU is defined as [17]:

$$\text{i-GELU}(x) = x \cdot \frac{1}{2} \left[1 + L \left(\frac{x}{\sqrt{2}} \right) \right] \quad (2.12)$$

$\text{clip}(|x|, \max = -b)$ is implemented as $\min(|x|, -b)$, and given that $x = qS$, where q is the quantized input and S is the scaling factor:

$$\min(|qS|, -b) = \min(S|q|, -b) = S \cdot \min(|q|, -\frac{b}{S}) \quad (2.13)$$

because $S > 0$ is a positive scalar parameter fixed for inference. Then we can rewrite Equation 2.11 as:

$$L(qS) = \text{sgn}(qS) [a(\min(S|q|, -b) + b)^2 + 1] \quad (2.14)$$

$$= \text{sgn}(q) \left[aS^2(\min(|q|, -\frac{b}{S}) + \frac{b}{S})^2 + 1 \right] \quad (2.15)$$

$$= aS^2 \cdot \text{sgn}(q) \left[\min(|q|, -\frac{b}{S})(\min(|q|, -\frac{b}{S}) + 2\frac{b}{S}) + \frac{b^2}{S^2} + \frac{1}{aS^2} \right] \quad (2.16)$$

$$= aS^2 \cdot \text{sgn}(q) [q_{\min}(q_{\min} + 2q_b) + q_c] \quad (2.17)$$

where $q_{\min} = \min(|q|, -\frac{b}{S})$, and $q_b = \frac{b}{S}$, and $q_c = \frac{b^2}{S^2} + \frac{1}{aS^2}$. Finally, we can implement the integer-only i-GELU function as:

$$\text{i-GELU}(qS) = qS \cdot \frac{1}{2} \left[1 + L\left(q\frac{S}{\sqrt{2}}\right) \right] \quad (2.18)$$

$$= qS \cdot \frac{1}{2} \left[1 + \frac{aS^2}{2} \cdot \text{sgn}(q) [q_{\min}(q_{\min} + 2q_b) + q_c] \right] \quad (2.19)$$

$$= \frac{aS^3}{4} \cdot q \left[\frac{2}{aS^2} + \text{sgn}(q) [q_{\min}(q_{\min} + 2q_b) + q_c] \right] \quad (2.20)$$

$$= \frac{aS^3}{4} \cdot q [q_1 + \text{sgn}(q) [q_{\min}(q_{\min} + 2q_b) + q_c]] \quad (2.21)$$

where $q_1 = \frac{2}{aS^2}$, and $\frac{aS^3}{4}$ can be considered as an output scaling factor. Note that q_b and q_c are calculated using $S/\sqrt{2}$ as the scaling factor. All the scaling factors are fixed once the training is done and nonlinear function coefficients are pre-computed for the inference.

Algorithm 1 shows the final integer-only implementation used for the hardware implementation of the i-GELU function. On line 6, there is a shift operation by constant k to avoid overflow. The values of q_1 and the output scaling factor are also updated due to this shift operation. The dataflow is shown in Figure 2.5.

Note that scaling factors might change when passing through a nonlinear function:

$$\text{GELU}(q_{\text{in}}S_{\text{in}}) = \text{i-GELU}(q_{\text{in}}) \cdot S_{\text{out}} \quad (2.22)$$

2.4.2 Softmax

Softmax normalizes an input vector $\mathbf{x}$ using the exponential function:

$$\text{Softmax}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^k \exp(x_j)}, \text{ where } \mathbf{x} = [x_1, \dots, x_k] \quad (2.23)$$

Algorithm 1 Integer-only GELU Approximation

```

1: function I-GELU( $q_{\text{in}}$ )
2:    $q_{\text{sgn}} \leftarrow \text{sign}(q_{\text{in}})$ 
3:    $q_{\text{min}} \leftarrow \min(|q_{\text{in}}|, -q_{\text{b}})$ 
4:    $q_{\text{L}} \leftarrow (q_{\text{min}} + 2q_{\text{b}})q_{\text{min}} + q_{\text{c}}$ 
5:    $q_{\text{erf}} \leftarrow q_{\text{sgn}} \cdot q_{\text{L}}$ 
6:    $q_{\text{erf}} \leftarrow q_{\text{erf}} \gg \gg k$ 
7:    $q_{\text{out}} \leftarrow q_{\text{in}} \cdot (q_{\text{erf}} + q_{\text{1}})$ 
8:   return  $q_{\text{out}}$ 
9: end function

```

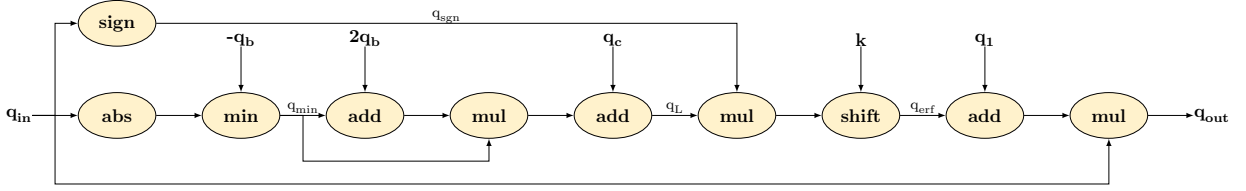


Figure 2.5: Dataflow of the i-GELU function

First, the maximum value is subtracted from each element to make the input to the exponential function a non-positive value, thus, limiting the output range to be within $[0, 1]$:

$$\text{Softmax}(\mathbf{x})_i = \frac{\exp(x_i - \max(x))}{\sum_{j=1}^k \exp(x_j - \max(x))} \quad (2.24)$$

The following decomposition is used to represent the non-positive real number $\tilde{x} = x_i - \max(x)$ [17]:

$$\tilde{x} = x_i - \max(x) = (-\ln 2)z + p \quad (2.25)$$

where z is a non-negative integer (quotient) and p is a real number in the interval $(-\ln 2, 0]$ (remainder). This decomposition enables efficient computation of the exponential function using integer arithmetic:

$$\exp(\tilde{x}) = \exp((-\ln 2)z + p) \quad (2.26)$$

$$= \exp(-\ln 2)^z \cdot \exp(p) \quad (2.27)$$

$$= (1/2)^z \cdot \exp(p) \quad (2.28)$$

$$= \exp(p) \gg \gg z \quad (2.29)$$

The value of $\exp(p)$ in the reduced range can be approximated using a second-order polynomial [17]:

$$L(p) = a(p + b)^2 + c \quad (2.30)$$

where $a = 0.3585$, $b = 1.353$, and $c = 0.344$. Replacing $\exp(p)$ with $L(p)$ in Equation 2.29, we get the integer-only approximation i-exp [17]:

$$\text{i-exp}(\tilde{x}) := L(p) \gg \gg z \quad (2.31)$$

where $z = \lfloor -\tilde{x} / \ln 2 \rfloor$ and $p = \tilde{x} + z \ln 2$. This can be calculated with integer arithmetic. Given that $\tilde{x} = qS$, where q is the quantized input and S is the scaling factor, we can rewrite z :

$$z = \left\lfloor -q \frac{S}{\ln 2} \right\rfloor \quad (2.32)$$

$$= \lfloor -q \left(\frac{S}{\ln 2} \cdot 2^{\text{fp_bits}} \right) \cdot 2^{-\text{fp_bits}} \rfloor \quad (2.33)$$

$$= -q \cdot \left(\frac{S}{\ln 2} \ll \text{fp_bits} \right) \gg \gg \text{fp_bits} \quad (2.34)$$

$$= -q \cdot q_{\ln 2, \text{inv}} \gg \gg \text{fp_bits} \quad (2.35)$$

where the real-valued ratio $\frac{S}{\ln 2}$ is approximated similarly as in Equation 4.6:

$$\frac{S}{\ln 2} = \frac{q_{\ln 2, \text{inv}}}{2^{\text{fp_bits}}} \quad (2.36)$$

$$q_{\ln 2, \text{inv}} = \frac{S}{\ln 2} \ll \text{fp_bits} \quad (2.37)$$

and the value of fp_bits is chosen as 30 for the inference which is sufficiently large to preserve accuracy while maintaining the coefficients in INT32 precision.

Note that right arithmetic shift essentially replaces floor operation. Similarly, for $\tilde{x} = qS$, the remainder p is computed as:

$$p = qS + z \ln 2 \quad (2.38)$$

$$= S \left(q + z \frac{\ln 2}{S} \right) \quad (2.39)$$

$$= S (q + z \cdot q_{\ln 2}) \quad (2.40)$$

$$= Sq_p \quad (2.41)$$

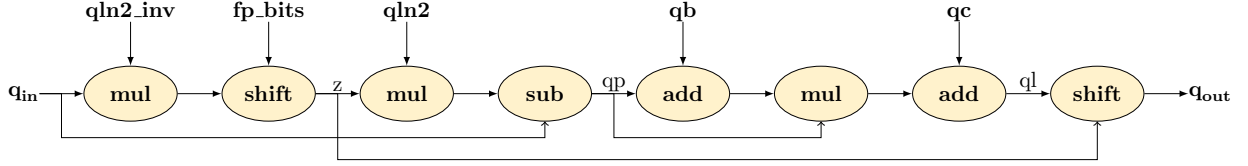


Figure 2.6: Dataflow of the i-exp function

where $q_{\ln 2} = \frac{\ln 2}{S}$, and $q_p = q + z \cdot q_{\ln 2}$. Similar to $L(x)$ in Section 2.4.1, when $p = q_p S$, $L(p)$ is expanded as:

$$L(p) = L(q_p S) = a(q_p S + b)^2 + c \quad (2.42)$$

$$= aS^2 \left[q_p \left(q_p + 2\frac{b}{S} \right) + \frac{b^2}{S^2} + \frac{c}{aS^2} \right] \quad (2.43)$$

$$= aS^2 [q_p(q_p + q_b) + q_c] \quad (2.44)$$

where $q_b = 2\frac{b}{S}$ and $q_c = \frac{b^2}{S^2} + \frac{c}{aS^2}$. Then, i-exp can be computed as:

$$\text{i-exp}(\tilde{x}) = L(p) >>> z \quad (2.45)$$

$$= aS^2 [q_l >>> z] \quad (2.46)$$

where $q_l = q_p(q_p + q_b) + q_c$. Output scaling factor is aS^2 , which can be dropped since a and S parameters are the same for all input vector elements and scaling factor gets canceled out in the softmax normalization as can be seen in Equation 2.24. Final i-exp implementation is shown in Algorithm 2.

Algorithm 2 Integer-only Exponential Approximation

```

1: function I-EXP( $q_{\text{in}}$ )
2:    $z \leftarrow -q_{\text{in}} \cdot q_{\ln 2.\text{inv}} >>> \text{fp\_bits}$ 
3:    $q_p \leftarrow q_{\text{in}} + z \cdot q_{\ln 2}$ 
4:    $q_l \leftarrow (q_p + q_b) \cdot q_p + q_c$ 
5:    $q_{\text{out}} \leftarrow q_l >>> z$ 
6:   return  $q_{\text{out}}$ 
7: end function

```

Algorithm 3 shows the integer-only implementation of the Softmax function. On line 4, q_{exp} is the output of i-exp function represented in INT32. On line 5, q_{req} is the INT16 result of requantizing q_{exp} with S_{req} and fp_bits following Equation 4.7. On line 7, factor is

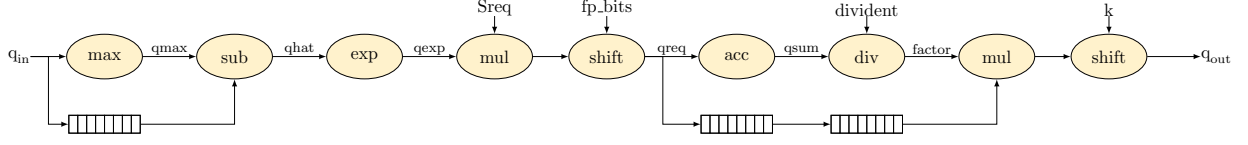


Figure 2.7: Dataflow of i-Softmax function

computed as an unsigned 32-bit integer division of dividend by q_{sum} , where dividend is a fixed constant value of 2^{30} and $q_{\text{sum}} > 0$. The constant 2^{30} is chosen to retain computations within INT32 precision while providing sufficient dynamic range for q_{sum} . In this way, we compute the factor once and use it for all the elements of the input vector, as shown in line 8. Shifting the result by $k = 30 - \text{out_bits}$ bits compensates for the multiplication by dividend from line 7 and ensures that the final result is in $\text{out_bits} = 8$ bits.

Algorithm 3 Integer-only Softmax Approximation

```

1: function I-SOFTMAX( $q_{\text{in}}$ )
2:    $q_{\text{max}} \leftarrow \max(q_{\text{in}})$ 
3:    $q_{\text{hat}} \leftarrow q_{\text{in}} - q_{\text{max}}$ 
4:    $q_{\text{exp}} \leftarrow \text{i-exp}(q_{\text{hat}})$ 
5:    $q_{\text{req}} \leftarrow q_{\text{exp}} \cdot S_{\text{req}} \gg \text{fp\_bits}$ 
6:    $q_{\text{sum}} \leftarrow \sum q_{\text{req}}$ 
7:    $\text{factor} \leftarrow \lfloor \text{divident} / q_{\text{sum}} \rfloor$ 
8:    $q_{\text{out}} \leftarrow (q_{\text{req}} \cdot \text{factor}) \gg k$ 
9:   return  $q_{\text{out}}$ 
10: end function

```

2.4.3 Layer Normalization

Layer Normalization standardizes the input values across the feature dimension, i.e., each row vector $\mathbf{x}$ of the input matrix, followed by weight scaling and bias addition:

$$\text{LayerNorm}(\mathbf{x}_i) = \frac{\mathbf{x}_i - \mu}{\sigma} \gamma + \beta \quad (2.47)$$

where μ is the mean of an input vector of size N , and σ^2 is the variance, computed as:

$$\mu = \frac{1}{N} \sum_{i=1}^N x_i \quad (2.48)$$

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2 \quad (2.49)$$

We observed that the variance calculation in Equation 2.49 needs two sequential passes over the input vector: the first pass to find the mean, and the second pass uses the mean to compute the variance. We optimize the variance computation to just a single pass by reformulating the Equation 2.49 as:

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2 \quad (2.50)$$

$$= \frac{1}{N} \sum_{i=1}^N (x_i^2 - 2x_i\mu + \mu^2) \quad (2.51)$$

$$= \frac{1}{N} \left[\sum_{i=1}^N x_i^2 - 2\mu \sum_{i=1}^N x_i + N\mu^2 \right] \quad (2.52)$$

$$= \frac{1}{N} \left[\sum_{i=1}^N x_i^2 - 2N\mu^2 + N\mu^2 \right] \quad (2.53)$$

$$= \frac{1}{N} \left[\sum_{i=1}^N x_i^2 - N\mu^2 \right] \quad (2.54)$$

$$= \frac{1}{N} \left[\sum_{i=1}^N x_i^2 - \mu \sum_{i=1}^N x_i \right] \quad (2.55)$$

In this way, we can compute the sum of the input elements to get the mean and the sum of the squared input elements in one parallel pass instead of two sequential passes.

For $x = qS$, we can compute the mean as:

$$\mu = \frac{S}{N} \sum_{i=1}^N q_i = S \cdot q_{\text{mean}} \quad (2.56)$$

$$q_{\text{mean}} = \left[\left(\frac{2^{\text{fp_bits}}}{N} \right) \cdot \sum_{i=1}^N q_i \right] \cdot 2^{-\text{fp_bits}} \quad (2.57)$$

where division by a known parameter N implemented as a multiplication by its inverse. Similarly, for variance calculation:

$$\sigma^2 = \frac{S^2}{N} \left[\sum_{i=1}^N q_i^2 - q_{\text{mean}} \sum_{i=1}^N q_i \right] = \frac{S^2}{N} \cdot q_{\text{var}} \quad (2.58)$$

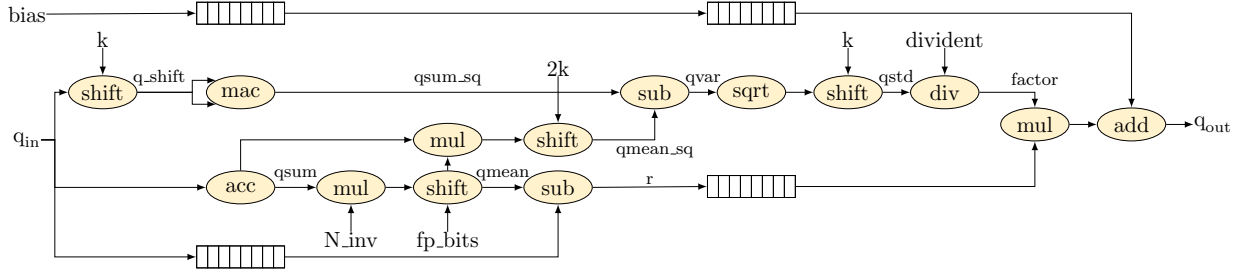


Figure 2.8: Dataflow of i-LayerNorm function

For a real valued $x = qS$, we can rewrite Equation 2.47 as:

$$\text{LayerNorm}(qS) = \frac{qS - \mu}{\sigma} \gamma + \beta \quad (2.59)$$

$$= \frac{qS - S \cdot q_{\text{mean}}}{\sqrt{\frac{S^2}{N} \cdot q_{\text{var}}}} \gamma + \beta \quad (2.60)$$

$$= \frac{S(q - q_{\text{mean}})}{\sqrt{\frac{S^2}{N} \cdot q_{\text{var}}}} \gamma + \beta \quad (2.61)$$

$$= \frac{q - q_{\text{mean}}}{\sqrt{q_{\text{var}}}} \sqrt{N} \gamma + \beta \quad (2.62)$$

$$= \sqrt{N} \gamma \left[\frac{q - q_{\text{mean}}}{\sqrt{q_{\text{var}}}} + \frac{\beta}{\sqrt{N} \gamma} \right] \quad (2.63)$$

$$= \sqrt{N} \gamma \left[\frac{q - q_{\text{mean}}}{\sqrt{q_{\text{var}}}} + q_{\beta} \right] \quad (2.64)$$

where $q_{\beta} = \frac{\beta}{\sqrt{N} \gamma}$, and $\sqrt{N} \gamma$ is the output scaling factor.

Algorithm 4 shows the integer-only implementation that uses bit shifts to handle scaling and prevent overflow. We shift the input by k bits to avoid overflow when accumulating the squared input elements, which is later compensated by shifting q_{std} by k bits. Similar to the Softmax strategy, instead of element-wise division by q_{std} , we compute a single factor and multiply it with the centered input r .

Algorithm 4 Integer-only Layer Normalization

```
1: function I-LAYERNORM( $q_{\text{in}}$ )
2:    $q_{\text{shift}} \leftarrow q_{\text{in}} \ggg k$ 
3:    $q_{\text{sq}} \leftarrow q_{\text{shift}} \cdot q_{\text{shift}}$ 
4:    $q_{\text{sum}} \leftarrow \sum q_{\text{in}}$ 
5:    $q_{\text{sum\_sq}} \leftarrow \sum q_{\text{sq}}$ 
6:    $q_{\text{mean}} \leftarrow q_{\text{sum}} \cdot N_{\text{inv}} \ggg \text{fp\_bits}$ 
7:    $r \leftarrow q_{\text{in}} - q_{\text{mean}}$ 
8:    $q_{\text{mean\_mul}} \leftarrow q_{\text{mean}} \cdot q_{\text{sum}}$ 
9:    $q_{\text{mean\_sq}} \leftarrow q_{\text{mean\_mul}} \ggg 2k$ 
10:   $q_{\text{var}} \leftarrow q_{\text{sum\_sq}} - q_{\text{mean\_sq}}$ 
11:   $q_{\text{std}} \leftarrow (\lfloor \sqrt{q_{\text{var}}} \rfloor) \lll k$ 
12:   $\text{factor} \leftarrow \lfloor \text{divident} / q_{\text{std}} \rfloor$ 
13:   $q_{\text{out}} \leftarrow (r \cdot \text{factor}) + q_{\beta}$ 
14:  return  $q_{\text{out}}$ 
15: end function
```

Chapter 3

Related Work

This section reviews recent efforts in accelerating transformer-based models on FPGAs. It covers architectural strategies for mapping transformers to reconfigurable hardware, and the various methods used for hardware implementation of nonlinear functions like Softmax, GELU, and Layer Normalization.

3.1 Transformer Quantization

Bhandare et al. [3] proposed a post-training quantization method for the pre-trained Transformer model [33], reducing weights and activations to 8-bit integers while preserving accuracy. Nonlinear functions were retained in 32-bit floating-point precision, as lower precision was argued to significantly degrade performance. Zafrir et al. [40] extended the work in [3] by introducing quantization-aware training for the BERT models, achieving 4x compression with minimal accuracy loss. Similar to [3], they kept nonlinear operations in FP32 precision but quantized both weights and activations of all linear layers to INT8 during the fine-tuning phase. This approach allowed the model to learn to compensate for quantization errors during training.

Mixed-precision quantization was used in [1], [41], [20], [30], and [19] to accelerate Transformer models on FPGAs. They quantize weights to INT4/6/8 but keep activations in 16-bit fixed-point representation or FP16. In contrast, fixed-point representation was used throughout the design in both [19] and [16].

I-BERT [17] applies similar quantization-aware training strategy and symmetric quantization scheme as in [40] for the pre-trained RoBERTa model [24], but keeps integer

precision throughout the design. Several works deploy integer-only inference. SwiftTron [25] implements I-BERT in a 65nm CMOS technology node using the ASIC design flow with Synopsys Design Compiler. I-ViT [23] applies integer-only methodology to Vision Transformers through a GPU-based implementation.

3.2 FPGA-based Transformer Accelerators

Often, nonlinear operations in FPGA-based neural networks are offloaded to CPUs with only matrix multiplications being accelerated. For example, Auto-ViT-Acc [22] offloads Softmax and GELU activations to the host CPU when accelerating Vision Transformers. They perform matrix multiplications on the FPGA using 3/4/8-bit fixed-point weights and activations. VAQF [30] also implements a Vision Transformer by decreasing the weight precision to binary (+1/-1) and using 6/8-bit integer activations. While they offload scaling, Softmax, and GELU activations to the host CPU, they maintain layer normalization on FPGA using 16-bit fixed-point precision.

Other works map nonlinear functions on-chip using fixed-point arithmetic. FTRANS [19] and NPE [16] both use 16-bit fixed-point representation and apply piecewise linear approximations for nonlinear functions. NPE [16] argues that lookup-based methods for nonlinear operations demand large and memory-intensive LUTs. LTrans-OPU [1] proposes a complex two-step approach to nonlinear functions: first training neural networks to approximate them, then convert these networks into piecewise linear functions implemented through 16-entry LUTs. While they use 8-bit quantization for weights and activations in matrix multiplications, their approach requires dequantization to fixed-point before nonlinear computations.

In some recent work, nonlinear functions are kept in floating-point precision on an FPGA. FlightLLM [41] and Pushing [20] accelerate LLaMA2-7B model on an FPGA by keeping nonlinear functions in FP16 precision. Zeng et al. [41] target both the Alveo U280 and the Versal VHK158, demonstrating improvements over NVIDIA GPUs in single-batch inference tasks. They compute matrix multiplications in INT8 precision. In contrast, Li et al. [20] deploys on the KV260 board, uses 4-bit weight quantization, and converts them to FP16 before multiplication. The outputs of matrix multiplications remain in FP16 for nonlinear functions. They employ on-chip INT8 quantization for some intermediate results to save memory.

Prior works on FPGA-based accelerators highlight the importance of always-on-chip inference schemes [19], [20], [41] and the resource-intensive nature of LUT-based approaches

[16], [1]. However, many designs lack crucial details about precision conversion between linear and nonlinear functions [20], [41] or omit detailed descriptions of their exponential function implementations [1], [20]. Furthermore, most hardware-focused works do not consider integration with quantization-aware training techniques [19], [16].

3.3 Nonlinear Functions

There are also LUT-based approaches to approximate nonlinear functions. Sun et al. [31] propose a VLSI architecture based on SMIC65nm technology for Softmax. Their “Basic-Split” approach decomposes exponential calculations into smaller basis components stored in LUTs in 32-bit fixed-point precision. Du et al. [11] target 65nm ASIC design and adopt a hybrid fixed-point approach that combines LUTs for exponential calculations and modified Maclaurin series with equal-interval LUTs for natural logarithm. Chen et al. [5] further optimize Softmax implementation using a hybrid method in 16-bit fixed-point precision: LUTs with exact restoring array dividers for the Most Significant Bits (MSBs), and Taylor expansion with logarithmic dividers for the Least Significant Bits (LSBs). They reduce resource utilization by more than 10 times compared to [31] with an increase in maximum absolute error from 0.399 to 0.447.

In contrast to fixed-point approaches, Belano et al. [2] implement Transformer-based models on ASIC in floating-point precision. Their SoftEx accelerator computes Softmax and GELU functions using floating-point exponential approximations. Their system operates with BF16 precision for inputs and outputs and uses FP32 for internal accumulation in Softmax. Whereas, I-ViT [23] implements all nonlinear functions (Shiftmax, ShiftGELU, I-LayerNorm) using integer arithmetic and shift-based operations, achieving a 3.7-4.1x speedup over FP32 models on GPUs.

Chapter 4

Implementation

This chapter describes the implementation details of the inference accelerator. We first describe the model’s computation flow in Section 4.1 to provide an overview of all the operations needed for inference. Then, we discuss each building block of the accelerator in the following sections using a bottom-up approach. Finally, we discuss the architecture design in Section 4.3 and optimizations made to reduce the resource utilization and latency in Section 4.4, as well as the batch processing approach for parallel sequence processing in Section 4.5.

4.1 Model Computation Flow

The model’s computation flow is shown in Algorithm 5, and the output matrix shapes are commented on each line, where N is the input sequence length, E is the embedding size, and V is the vocabulary size. Embedding and vocabulary sizes are model-specific parameters. For the I-BERT base model, $E = 768$ and $V = 50265$.

Algorithm 5 Model End-to-End Computation.

```
1: function MODEL( $X_{\text{tokens}}$ )  
2:    $X_{\text{embed}} \leftarrow \text{Embedding}(X_{\text{tokens}})$   $\triangleright (N, E)$   
3:    $X_{\text{hidden}} \leftarrow \text{Encoder}(X_{\text{embed}})$   $\triangleright (N, E)$   
4:    $X_{\text{logits}} \leftarrow \text{LanguageModelHead}(X_{\text{hidden}})$   $\triangleright (N, V)$   
5:   return  $X_{\text{logits}}$   
6: end function
```

For example, the following input sequence “Sun was shining, and <mask> were singing, and kids’ joy was filling the air. Dogs’ <mask> were echoing as the day’s work began.” has exactly 32 tokens: [0, 8816, 21, 21003, 6, 8, 50264, 58, 6970, 6, 8, 1159, 108, 5823, 21, 8794, 5, 935, 4, 18619, 108, 50264, 58, 29303, 25, 5, 183, 18, 173, 880, 4, 2]. Tokens 0 and 2 are special tokens that indicate the beginning and the end of the sequence, respectively. Token 8816 encodes the word “Sun”, 6 is for the comma punctuation mark “,”, 21 is for the word “was”, and so on. Token 50264 indicates the special masked token <mask>, which the model will predict by understanding the context around it in the sequence.

For this masking task, the LanguageModelHead is selected to be a linear layer that transforms the encoder output into a logits matrix where each row is of vocabulary size. Logits can be considered as the unnormalized probabilities of the vocabulary tokens for each position in the sequence. We apply argmax to X_{logits} to find the index with the largest value in each row of the matrix. These indices correspond to the predicted token values in our vocabulary. Finally, predicted integer tokens are decoded back into words using the model’s vocabulary, e.g., masked positions 7 and 22 in the above example sequence might be decoded back to “birds” and “voices”, respectively.

Algorithm 6 Encoder Computation

```

1: function ENCODER( $X_{\text{embed}}$ )
2:    $X_{\text{hidden}} \leftarrow X_{\text{embed}}$   $\triangleright (N, E)$ 
3:   for  $l \leftarrow 0$  to  $L - 1$  do
4:      $X_{\text{hidden}} \leftarrow \text{EncoderLayer}(X_{\text{hidden}})$   $\triangleright (N, E)$ 
5:   end for
6:   return  $X_{\text{hidden}}$ 
7: end function

```

In the model’s computation flow, shown in Algorithm 5, we accelerate the main Encoder function on the FPGA, as it is the most time-consuming part of the model (80-90% of measured runtime on both the CPU and GPU), and leave the rest of the processing steps, specifically the embedding and language model head, to the host CPU. The Encoder function is shown in Algorithm 6, where L is the number of layers, H is the number of attention heads, $d = E/H$ is the size of each attention head, and F is the hidden dimension size of the feed-forward networks. For the I-BERT base model, $L = 12$, $H = 12$, $d = 64$, $F = 3072$. We store the model parameters off-chip and supply them to the accelerator through the AXI interface. Input embeddings are also supplied through the AXI interface, and the Encoder output is received from the FPGA once the last layer is computed and passed to the LanguageModelHead on the host CPU. We send the weights only once and keep all the activations on-chip (except for the slicing case in Section 4.4.1).

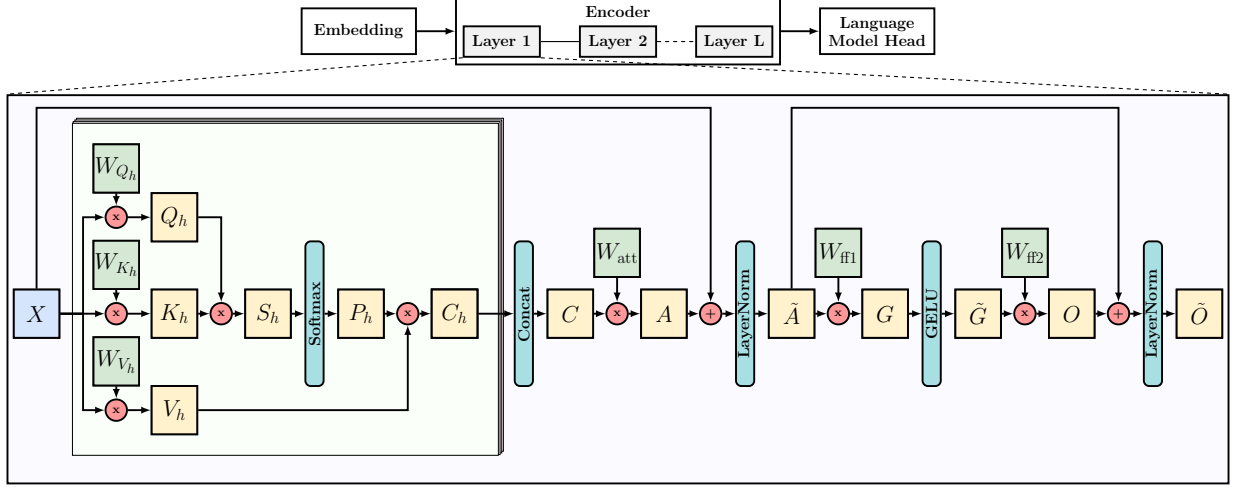


Figure 4.1: Computation flow of a BERT encoder layer.

Algorithm 7 Encoder Layer Computation

```

1: function ENCODERLAYER( $X$ )
2:   for  $h \leftarrow 0$  to  $H - 1$  do
3:      $Q_h \leftarrow XW_{Q_h} + b_{Q_h}$   $\triangleright (N, d)$ 
4:      $K_h \leftarrow XW_{K_h} + b_{K_h}$   $\triangleright (N, d)$ 
5:      $V_h \leftarrow XW_{V_h} + b_{V_h}$   $\triangleright (N, d)$ 
6:      $S_h \leftarrow Q_h K_h^T$   $\triangleright (N, N)$ 
7:      $P_h \leftarrow \text{softmax}(\frac{S_h}{\sqrt{d}})$   $\triangleright (N, N)$ 
8:      $C_h \leftarrow P_h V_h$   $\triangleright (N, d)$ 
9:   end for
10:   $C \leftarrow \text{Concat}(C_0, C_1, \dots, C_{H-1})$   $\triangleright (N, E)$ 
11:   $A \leftarrow CW_{\text{att}} + b_{\text{att}}$   $\triangleright (N, E)$ 
12:   $\tilde{A} \leftarrow \text{LayerNorm}(A + X)$   $\triangleright (N, E)$ 
13:   $G \leftarrow \tilde{A}W_{\text{ff1}} + b_{\text{ff1}}$   $\triangleright (N, F)$ 
14:   $\tilde{G} \leftarrow \text{GELU}(G)$   $\triangleright (N, F)$ 
15:   $O \leftarrow \tilde{G}W_{\text{ff2}} + b_{\text{ff2}}$   $\triangleright (N, E)$ 
16:   $\tilde{O} \leftarrow \text{LayerNorm}(O + \tilde{A})$   $\triangleright (N, E)$ 
17:  return  $\tilde{O}$ 
18: end function

```

Parameter	Dimension
X	(N, E)
$W_{Q_h}, W_{K_h}, W_{V_h}$	(E, d)
$b_{Q_h}, b_{K_h}, b_{V_h}$	$(1, d)$
W_{att}	(E, E)
b_{att}	$(1, E)$
W_{ff1}	(E, F)
b_{ff1}	$(1, F)$
W_{ff2}	(F, E)
b_{ff2}	$(1, E)$

Table 4.1: Dimensions of model input and parameters.

In Algorithm 7, the weights and biases of the EncoderLayer function are layer-specific. The layer index is omitted for brevity. The dimensions of the parameters are shown in Table 4.1:

4.2 Linear Layer

The first building block of our accelerator is a Linear layer that consists of matrix multiplication and bias addition:

$$Y = XW + b \quad (4.1)$$

where X is the input matrix, W is the weight matrix, and b is the bias vector.

Matrix multiplication is implemented using a systolic array, and bias vector addition is done within the subsequent requantization unit.

4.2.1 Systolic Array

A systolic array is a 2-dimensional grid of processing elements (PEs) connected to each other in a nearest-neighbor fashion. The array has a parameterizable dimension of $N_1 \times N_2$, which enables scalability to larger designs. Each PE performs a multiply and accumulate (MAC) operation on a pair of elements from the input matrices, essentially performing a dot product on a row of the first input matrix X and a column of the second input matrix W .

Input matrices X and W are stored in $N1$ and $N2$ memory banks respectively, such that the systolic array can read $N1$ rows of X (one element per row) and $N2$ columns of W (one element per column) in parallel every cycle. New elements from X and W are supplied to the leftmost column and topmost row of the array respectively. Each PE gets a new pair of X and W every cycle and passes them to the right and bottom PEs respectively. The W matrix is vertically split into column-wise blocks, and stored in double-buffered memory banks to overlap the transfer of the weight matrix with the block multiplication. Once a PE completes a dot product, the accumulator is reset and the result propagates horizontally to the right until the rightmost PE, where it gets written into the output memory bank.

The systolic array is a data-flow architecture that avoids frequent memory accesses and enhances the reuse of the input data. It is a highly parallel design where computations are performed simultaneously on all PEs. Data movement is local, which reduces routing costs and improves timing.

The systolic array design is mainly adapted from the ECE327 Digital Hardware Systems course laboratories of the University of Waterloo.

4.2.2 Requantization

The systolic array receives quantized inputs in INT8 precision and accumulates the results in INT32 precision for better accuracy. Requantization performs conversion of the accumulator data type to match the input precision for the next matrix multiplication, e.g., the accumulator is converted from INT32 to INT8. We fuse bias addition with the requantization process since they are both vector operations applied to a matrix row.

Given an integer q_{32} and its scaling factor S_{32} , from Equation 2.1, we can express the real value x_{32} as:

$$x_{32} = q_{32}S_{32} \quad (4.2)$$

Similarly, given an integer q_8 and its scaling factor S_8 , we can express the real value x_8 as:

$$x_8 = q_8S_8 \quad (4.3)$$

When changing the precision of an integer, we need to ensure that the real value remains the same: $x_{32} = x_8$. Therefore, we can derive the following equation:

$$q_{32}S_{32} = q_8S_8 \quad (4.4)$$

$$q_8 = \frac{S_{32}}{S_8}q_{32} \quad (4.5)$$

The ratio of real-valued scaling factors is approximated using the Dyadic Numbers concept [32]:

$$\frac{S_{32}}{S_8} = \frac{a}{2^b} \quad (4.6)$$

where a and b are integers. Thus, requantization can be implemented using integer addition, multiplication, and bit shift operations:

$$Y_{i,j} = \text{Clamp}(((X_{i,j} + b_j) \times m_j) \gg r_j, -M, M) \quad (4.7)$$

where $X_{i,j}$ is the input matrix element at row i and column j ; b_j , m_j , and r_j are the elements of the bias, requantization multiplier, and shift amount vectors, respectively; M is the maximum representable value in the output precision; and $\gg$ denotes the arithmetic right-shift operation. Requantization module is fully pipelined and can accept an input every cycle. All the vectors are stored on-chip and are read fully N times, where N is the number of rows in the input matrix.

4.3 Accelerator Architecture

We construct the Encoder layer from Figure 4.1 in Section 4.3.4 by first dividing it into three main modules: Attention Head in Section 4.3.1, Linear Layer followed by Layer Normalization in Section 4.3.2, Linear Layer followed by GELU in Section 4.3.3.

4.3.1 Attention Head

The Attention Head module implements the self-attention mechanism described in Section 2.1.2. The input tensor $X \in \mathbb{Z}^{N \times E}$ undergoes three parallel matrix multiplications with the weight matrices W_{Q_h} , W_{K_h} , and W_{V_h} (each $\in \mathbb{Z}^{E \times d}$), implemented using systolic arrays, where N is the number of tokens, E is the embedding dimension, and $d = E/H$ is the dimension of the attention head, and H is the number of heads. These multiplications produce INT32 results Q_h , K_h , and V_h (each $\in \mathbb{Z}^{N \times d}$). To pass the results into the next matrix multiplication, we first requantize them back to INT8 representations Q'_h , K'_h , and V'_h . The attention context C_h , which is the result of multiplying P_h and V'_h , also undergoes a requantization from INT32 to INT8. Note that the Softmax accepts INT32 input and produces INT8 output.

The `attn.head` module employs a total of five systolic array multipliers, visualized as red multiplication circles in Figure 4.2. Each of the systolic arrays can be configured

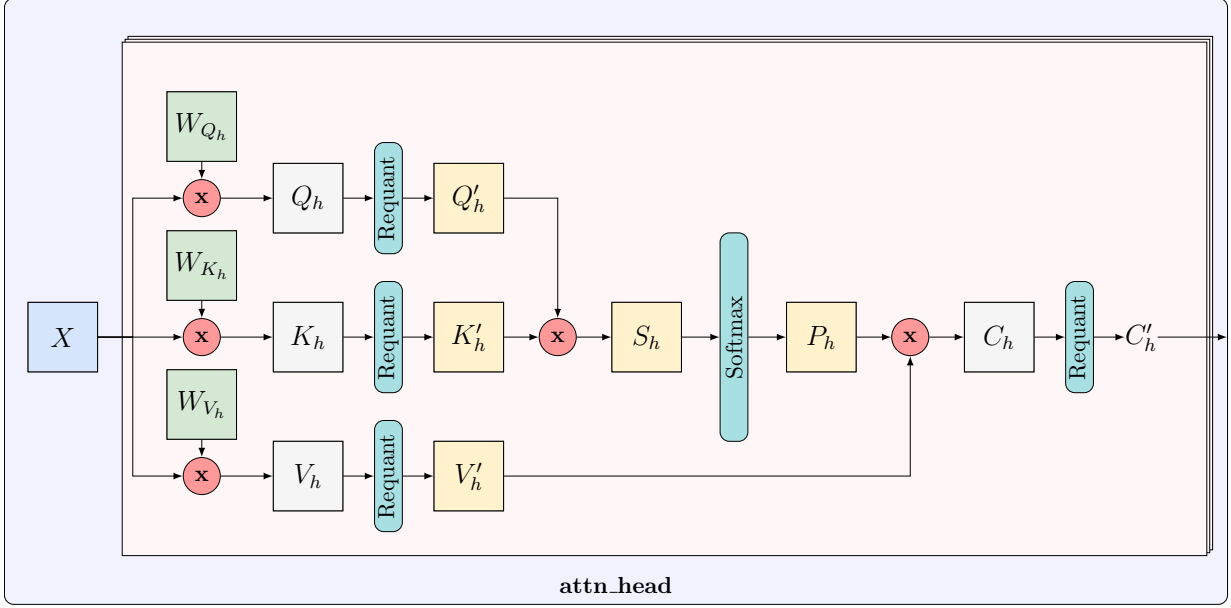


Figure 4.2: Dataflow of the `attn_head` module

to unique $N1 \times N2$ dimensions depending on the input matrix sizes. The outputs of the systolic arrays are stored in memory banks. Once the multiplications are finished, the results are streamed to the next module and systolic arrays can accept the next set of inputs. Systolic array output is stored in $N1$ memory banks and is transferred from parallel banks to serial streams. The Softmax module is fully pipelined and can accept a new input every cycle. It accepts input vectors (S_h matrix rows) as a stream, one element per cycle, and produces an output stream of one element per cycle.

4.3.2 Linear Layer and Layer Normalization

The `mm_ln` module, shown in Figure 4.3, performs a linear transformation and a requantization followed by a residual addition and a Layer Normalization. There are two instances of the `mm_ln` module in the encoder layer: one for the multi-head self-attention output processing (Self Output) and one for the feed-forward network (Layer Output) shown on lines 10-12 and 15-16 of Algorithm 6, respectively.

The residual input is stored in row-major order in a circular buffer where it can be written while getting read. Matrix addition is performed as a simple adder module with two streaming inputs. Similar to the Softmax module, the Layer Normalization module is

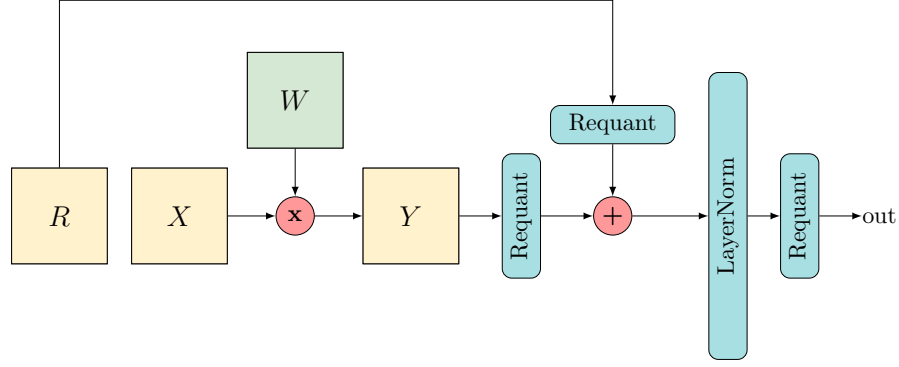


Figure 4.3: Dataflow of the `mm_ln` module

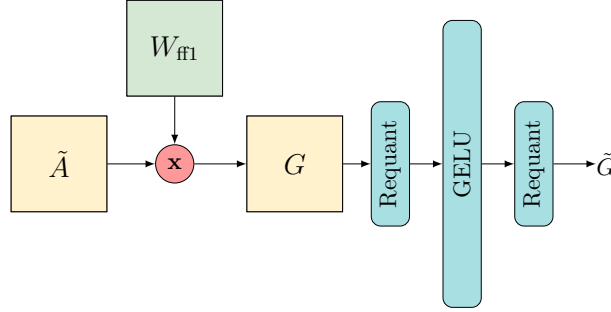


Figure 4.4: Dataflow of the `mm_gelu` module

also fully pipelined and can accept a new input every cycle. It accepts an input stream from the adder and produces an output stream that goes through a requantization and then to the next stage.

4.3.3 Linear Layer and GELU

The `mm_gelu` module, shown in Figure 4.4, performs a linear transformation and applies a GELU function followed by a requantization. Here, the requantization after a matrix multiplication is just for a bias addition. The GELU module is fully pipelined consumes an INT32 input stream and produces an INT8 output stream that goes through requantization to INT8.

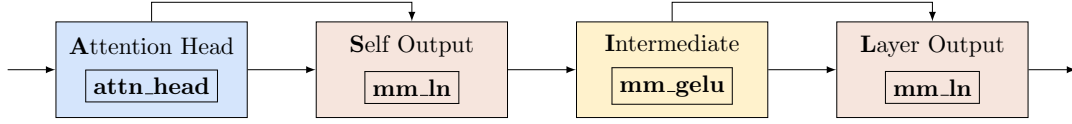


Figure 4.5: Dataflow of the **encoder** module

4.3.4 Encoder

Figure 4.5 shows the **encoder** module that stitches all the instances of the **attn_head**, **mm_ln**, and **mm_gelu** modules. It consists of four modules: Attention Head, Self Output, Intermediate, and Layer Output. It handles the external AXI4-Stream interface: supplies the input stream to the corresponding modules (by demultiplexing it into four modules if there is only a single channel AXI) and produces the final output stream after iterating through all the layers.

4.4 Architecture Optimizations

The following Subsections 4.4.1, 4.4.2, 4.4.3, and 4.4.4 describe the optimizations applied to the architecture where only a single AXI input port is available to send the weights stored in the off-chip memory to the FPGA. In Section 4.5, we discuss the design with 4 AXI input ports to perform pipelined batch processing.

4.4.1 Slicing

The initial design without all the optimizations was aimed at the case where all of the activations could not be kept on-chip and had to be written back to off-chip memory. There are two major memory consumers in the encoder layer: activations and weights. Since the weight matrices were already vertically split into column-wise blocks (also discussed in Section 4.4.2) and we store only two blocks of weights in double-buffered fashion (adopted from the ECE327 labs), the optimization was targeted at splitting the activations. The idea was to horizontally split the activations with initial dimensions of (D_1, D_2) into slices of $(D_1/S, D_2)$, where S is the number of slices. When multiplied with a weight matrix of dimensions (D_2, D_3) , the result is a matrix of dimensions $(D_1/S, D_3)$. Splits for the activations are done horizontally, since a matrix multiplication requires a full row of an activation matrix and a full column of a weight matrix to compute a single element of the output matrix. Figure 4.1 shows that **attn_head** contains a multiplication between

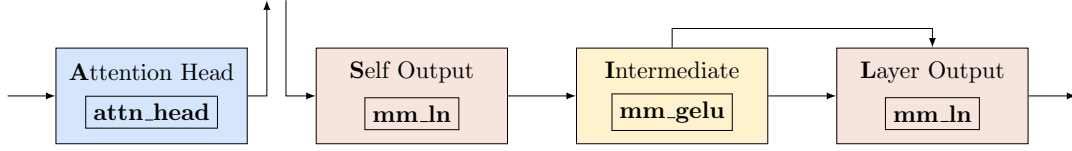


Figure 4.6: Dataflow of the Slicing optimization

two activations: QK^T . By applying slicing, QK^T would result in a matrix of dimensions $(D_1/S, D_1/S)$. However, Softmax requires a full row of QK^T due to the reduction operation performed on matrix rows. Thus, we apply slicing only to the `mm_ln` and `mm_gelu` modules. So, the lines 10-16 of Algorithm 6 were performed on activations where the first dimension was sliced to N/S . The slicing strategy needs one layer’s weights to be sent S times to the encoder except for `attn_head` where it was sent only once.

Figure 4.6 shows how the inputs to modules are connected with respect to the AXI interface. First, the input embeddings and attention weights are supplied to `attn_head` without slicing. Then, the `attn_head` output is written back to off-chip memory, since the following modules operate on just a single slice of the activations. Once the `attn_head` finishes computing all the heads, its outputs C_h are concatenated and sliced to be sent to the next module. Now the rest of the encoder layer executes one slice of the activations at a time. Note that the sliced activation is provided from the off-chip memory only to the Self Output module and the next modules receive their input data from the output of the previous modules following the dataflow shown in Figure 4.6.

4.4.2 One Systolic

From Figure 4.2, we can see that the three systolic arrays that compute Q_h , K_h , and V_h can operate in parallel if they have input X and all the weights W_{Q_h} , W_{K_h} , and W_{V_h} available. However, to save memory usage, we store only two blocks of weights for a systolic array in a double-buffered fashion. One block of weights has a dimension of $(D_1, D_2/B)$, where B is the number of blocks. This is done to overlap the weight transfer with the matrix multiplication. Thus, one AXI input port is exclusively occupied with weight transfer to just one systolic array while the multiplication is performed. We also strategically choose the systolic dimensions such that AXI interface is kept idle as little as possible. Exclusive ownership of AXI port by one systolic array during multiplication limits the parallel execution of multiple systolic. On the boards, where we have only one AXI input port, `attn_head` has only one of the Q_h , K_h , and V_h systolics active at a time. This means we can replace the three systolic arrays in `attn_head` with one systolic array to compute

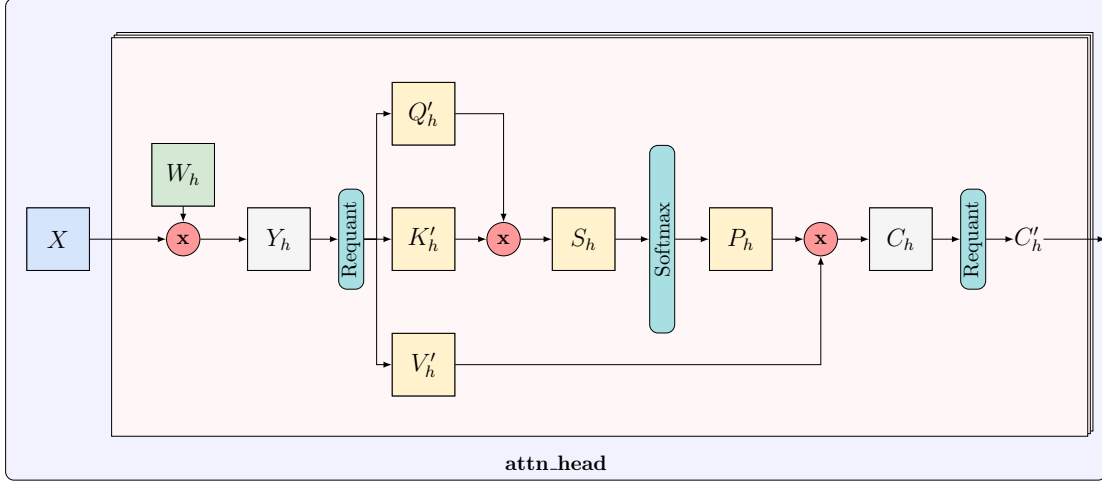


Figure 4.7: Dataflow of the One Systolic optimization

Q_h , K_h , and V_h sequentially and save on the valuable resources without any performance loss. Note that this applies to a systolic array that receive one of the inputs from the off-chip memory through the AXI interface, and does not affect the systolic computing S_h and P_h that have both inputs coming from activations stored on-chip. The output of the QKV systolic array is then multiplexed to the next systolics as shown in Figure 4.7.

4.4.3 Read Residuals

From the Figures 4.1 and 4.3, we can see that `mm_ln` modules need to read the residuals from one of the previous inputs to perform addition and Layer Normalization. In Algorithm 6, these residuals are denoted as X and $\tilde{A}$ on lines 12 and 16 for the first and second `mm_ln` modules respectively. In a fully pipelined design, residuals are stored in dedicated circular buffers and supplied directly when X and $\tilde{A}$ matrices are streamed. Residual X is taken from the AXI interface at the same time when it gets written to `attn_head` systolic memories. Residual $\tilde{A}$ is taken from the output of the previous `mm_ln` module when it is streamed to the `mm_gelu` module. Similar to the one systolic optimization, these extra buffers dedicated to residuals can be removed since there is only one AXI input port and only one systolic can be active in the whole encoder datapath. Circular buffers would be needed for the batched design where all the modules can be active at the same time.

Since we do not store residuals separately, they need to be read from the systolic memories after their matrix multiplications are completed. Residuals X and $\tilde{A}$ are taken from the

activation memories of the systolics in the `attn_head` and `mm_gelu` modules respectively.

4.4.4 Block-wise Output

All the systolic arrays fully store the matrix multiplication output in the output memory banks. Since the weights are supplied block-wise, the systolic array writes only one block of the output at a time. Therefore, there is a possibility to save on memory usage by allocating systolic output memories to store only one block of the output and immediately stream it to the next module to free up space for the next block. However, this optimization does not apply to systolic that supply output to nonlinear functions that need to see the whole row of the matrix. Such nonlinear functions are Softmax and Layer Normalization as they contain vector reduction operations performed on matrix rows. Fortunately, the GELU function is an element-wise operation and does not need the whole row of the input matrix to produce an output. Therefore, we can save on the memory usage by storing only one block of the output $G = XW_{\text{ff1}}$. This optimization gives us a significant reduction in memory utilization since the G matrix is the largest activation in the encoder layer. Requantization modules in Figure 4.4 are adjusted to iterate every block size instead of the whole row size.

4.4.5 AXI Packing

Data is transferred from the off-chip memory to the FPGA through the AXI4-Stream interface. Almost all of the transfers are for sending the weights. For the IBERT-base model, the number of elements in the weights matrices is more than 99% of the total number of parameters including biases, requantization vectors, etc. Therefore, it is crucial to speed up the weight transfer. After quantization, the weights are represented in 8-bit precision. We deploy a packing technique to pack the weights into the AXI streams that can be wider than weights precision, such as 32-bit, 64-bit, 128-bit, or more. We fully utilize the AXI bandwidth by pre-packing the weights and sending them in parallel. On the PL side, we unpack the weights of the systolic arrays. Wide transfers increase the complexity of the memories that store the weights which is addressed in subsection 4.6.1.

4.5 Batch Processing

The pipelined architecture of our accelerator is used to its full potential when a batch of multiple sequences are processed in parallel. This is achieved by having four AXI ports

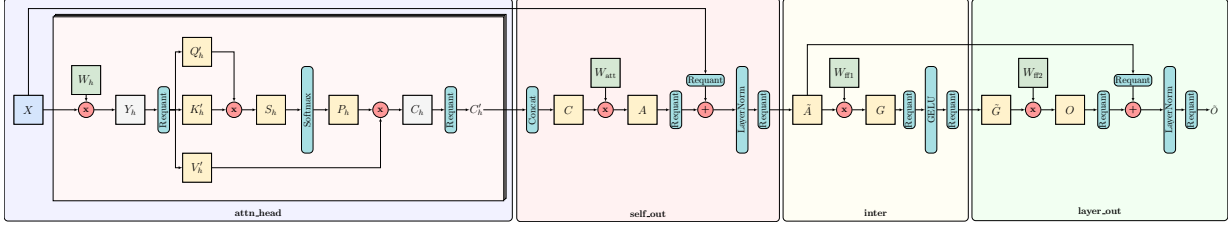


Figure 4.8: Pipelined batch processing

to send data from the Processing System (PS) to the Programmable Logic (PL) since the encoder layer is divided into four pipeline stages: **A**ttention Head - **attn_head**, **S**elf Attention Output - **self_out**, **I**ntermediate - **inter**, and **L**ayer Output - **layer_out**. This is shown in Figure 4.8. Each of these stages has a systolic array that performs matrix multiplication of activation with a weight matrix, shown as green rectangles in Figure 4.8. From the discussion in Subsection 4.4.2, we know that each systolic array, that has an operand as a weight matrix coming from the PS, takes exclusive ownership of one AXI port during multiplication. Therefore, we need four AXI ports to have all the systolic arrays active at the same time. Thus, the upper limit of the batch size is 4 which enables the maximum throughput of the accelerator.

4.5.1 Pipelined Execution

Tables 4.2 and 4.3 illustrate the execution patterns for sequential and pipelined processing respectively. In the sequential execution, each sequence (S0-S3) completes all its modules before the next sequence begins. In contrast, the pipelined execution allows modules to process different sequences simultaneously, significantly reducing the total processing time. The reduction in the total processing time is less than 4 times because of the three starting (1-3) and three finishing (49-51) computation cycles where the pipeline is not fully filled. The timing diagram for each module's states is given in Figure 4.10, where (a) shows the fully sequential design and (b) shows the perfectly balanced pipelined design with batch processing. We can see that all four stages are active simultaneously in the pipelined design with batching enabled.

4.5.2 Pipeline Buffers

Simultaneous execution of the four stages requires separate buffers for residuals. The reason is that the residuals are consumed only after the matrix multiplication is completed. This

	1	...	48	49	...	96	97	...	144	145	...	192	
S0	A	S	I	L									
S1				A	S	I	L						
S2							A	S	I	L			
S3										A	S	I	L

Table 4.2: Sequential execution pattern showing each sequence (S0-S3) completing all modules 12 times before the next sequence begins. A, S, I, and L represent Attention Head, Self Attention Output, Intermediate, and Layer Output modules respectively.

	1	2	3	4	5	...	48	49	50	51	
S0	A	S	I	L	A	S	I	L			
S1		A	S	I	L	A	S	I	L		
S2			A	S	I	L	A	S	I	L	
S3				A	S	I	L	A	S	I	L

Table 4.3: Pipelined execution pattern showing concurrent processing of different sequences across modules. Each sequence (S0-S3) flows through the pipeline while allowing multiple modules to be active simultaneously.

means that the previous module, e.g., `attn_head`, has to wait until the next module, e.g., `self_out`, consumes the residual. This results in `attn_head` being idle the whole time when `self_out` is performing multiplication. This is illustrated in Figure 4.10-c, where `attn_head` always waits for `self_out` to finish the current sequence to start a new one. `attn_head` keeps the residual in its systolic memory until `self_out` consumes it, which happens almost at the end of `self_out` stage. A similar situation occurs with `inter`, which supplies residual to `layer_out` stage. Thus, we put separate buffer storages, as shown in the Figure 4.9, for residuals to avoid these wasted wait cycles.

We also need to place extra buffers at the outputs of each stage to avoid one module being idle while it has finished its execution but the next module is not yet ready to receive

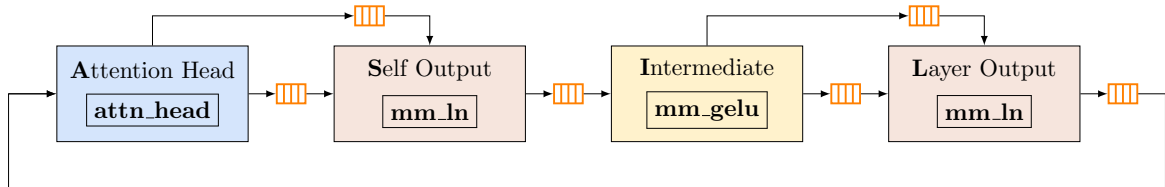


Figure 4.9: Encoder layer with residual and output buffers. Orange boxes represent buffers.

input. This is illustrated in Figure 4.10-d, where `inter` keeps the output of sequence S1, waiting for `layer_out` to finish processing the previous sequence S0. This waiting propagates back to `self_out` stage, which also has to wait until `inter` sends its output and can receive a new sequence of S2. To reduce the waiting time due to unbalanced execution times, we need to buffer the outputs of each stage, so each module can start processing the next sequence without waiting for its outputs to be consumed for at least one more sequence. Output buffers are also shown in the Figure 4.9.

Although output buffers seem to be mainly required by the unbalanced pipeline stages, there is also a potential deadlock situation where temporary storage is needed. For example, in the perfectly balanced pipeline shown in Figure 4.10-b, with a batch size of 4, once `layer_out` completes processing sequence S0, its output has nowhere to go because `attn_head` is not yet ready to accept new data. `attn_head` is not ready since its output after processing sequence S3 was not yet consumed by `self_out`, which is still busy because of sequence S2, and `inter` is busy because of sequence S1. To resolve this deadlock caused by circular dependency, the output of `layer_out` needs to be stored temporarily in a buffer.

Residual and output buffers are circular buffers since they are written and read only once as a stream in sequential order. Both of these buffers need just one read and one write port. Therefore, we use optimized URAMs, discussed in Subsection 4.6.1, to instantiate these circular buffers. The depth of the buffers is equal to the input matrix size to accommodate activations of at least one full sequence.

4.6 FPGA Optimizations

In this section, we discuss the optimizations applied to the FPGA implementation.

4.6.1 Memory Optimizations

The efficient utilization of on-chip memory resources is crucial to be able to fit all the encoder layer activations on-chip. When synthesizing memories of arbitrary dimensions, the default Vivado synthesis tool mapping to Block RAMs (BRAMs) and UltraRAMs (URAMs) can lead to suboptimal resource usage.

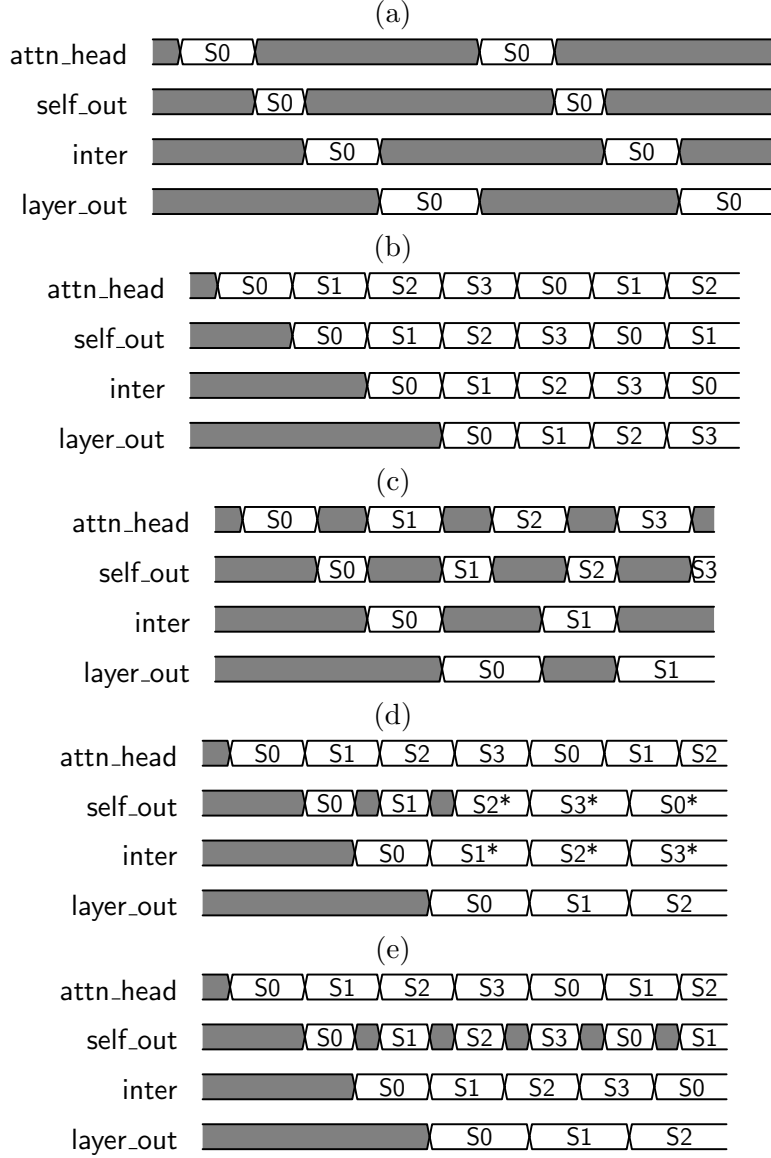


Figure 4.10: Execution Waveforms: (a) fully sequential design, (b) perfectly balanced pipelined design with batch processing, (c) batch processing with wasted cycles due to absence of residual and output buffers, (d) batch processing with residual buffers, but without output buffers, (e) batch processing with residual and output buffers

BRAM Instantiation

BRAM36 is a 36K-bit memory primitive and BRAM18 is an 18K-bit memory primitive. BRAMs can be configured with different depths and widths. BRAM36 can be decomposed into two BRAM18s. To optimize memory resource utilization, we implemented custom memory mapping strategies. Instead of allowing the synthesis tool to automatically map memories, we explicitly partitioned them into more efficient configurations. For example, a naive instantiation of a 3K-depth 32-bit wide memory gets synthesized into 4 instances of BRAM36 primitives consuming more resources than strictly necessary. Which is a 4K-depth of 36-bit instead of the desired 3K-depth of 32-bit. We restructured the mapping to explicitly instantiate three BRAM36 primitives, each configured for 1K-depth and 32-bit width. We noticed that instantiating 32-bit wide memories is also possible by using BRAM18 primitives. The technique is to instantiate a memory with 512×32 -bit dimensions. Such finer mapping is beneficial since it reduces memory wastage compared to $1K \times 32$ -bit mapping. It covers the cases where the depth is not a multiple of 1K, e.g., for $1.5K \times 32$ -bit memory, we instantiate 3 BRAM18s instead of 2 BRAM36s. Similarly, for wider but shallower memories, such as those with $24K \times 8$ -bit dimensions, the default synthesis would map to 16 instances of BRAM18 primitives. Our approach remaps this to 12 instances of BRAM18 primitives, each configured for 2K-depth and 8-bit width, effectively storing 16K-bit data per BRAM18.

URAM Instantiation

URAM is a 288K-bit memory primitive and can be configured with a 4K-depth and a width of 72-bit. URAMs might not be present on all the FPGA devices. In our datapath, the best place to use URAMs is for deep circular buffers. For example, residual memory needs to store $24K \times 8$ -bit data which needs 192K-bit storage. The default mapping would be 16 instances of BRAM18s. With the BRAM optimizations, we can remap this to 12 instances of BRAM18s, each configured for 2K-depth and 8-bit width. When URAMs are available, we can save on the BRAMs by further remapping this to just a single instance of URAM. However, naively instantiating a $24K \times 8$ -bit memory synthesizes to 6 URAMs. Therefore, we deploy a packing technique to change the width to 64 bits. In this way, our $24K \times 8$ -bit memory is reconfigured to $3K \times 64$ -bit memory, synthesizing to just 1 URAM. To correctly perform the writes, we need to register the 8-bit input data for 8 cycles. Similarly, when we read from an address, we need to slice the correct 8-bit data from the 64-bit word.

Merge Double Buffers

We observed that instantiating double buffers for weights as two separate memories is not effective when the necessary depth is shallow. Therefore, we merge the double buffers into a single memory by logically splitting it into upper and lower address parts. For example, two buffers of 384×32 -bit are merged into a single 768×32 -bit memory, where the first buffer has an address range of 0-383 and the second buffer has an address range of 384-767. This requires separate read and write ports for the upper and lower parts. Although BRAMs can have true dual-port configurations, double buffering requires only single write and read ports since only one buffer is written at a time, and only one buffer is read at a time.

4.6.2 DSP Packing

We adopted the DSP packing technique from [12] to merge two adjacent PEs in the systolic array to save on the DSP resources. DSP48E1 and DSP48E2 are multiplier-accumulator primitives that can be configured as 25×18 -bit and 27×18 -bit multipliers, respectively, with a 48-bit accumulator. DSPs have a preadder that can add two numbers and produce a result that goes as a multiplier operand. Adder operands' precisions should be limited to 24 and 26 bits for E1 and E2, respectively, to prevent overflow or underflow. The DSP equation is given by:

$$P = (A + D) \cdot B \quad (4.8)$$

where A and D are preadder operands and B is the multiplier operand. To pack two INT8 multiplications into a single DSP, we need to reformulate the equation as:

$$P = A \cdot B + D \cdot B \quad (4.9)$$

where B would be the common multiplier. Equation 4.9 can be implemented in the systolic array to merge two vertically adjacent PEs that operate on the same column of the weight matrix and two rows of the activation matrix, or two horizontally adjacent PEs that operate on the same row of the activation matrix and two columns of the weight matrix.

We need to ensure that the preadder result contains A and D in the upper and lower bit positions respectively with enough separation such that the multiplier P can be correctly split into two different multiplication outputs. This is achieved by shifting the input A by at least 16 bits since $D \cdot B$ will at most occupy the lower 16 bits of P . It means that the preadder should be at least $16 + 8 = 24$ bits wide which is provided by both DSP types.

So, the final DSP packing equation is:

$$\{P_1, P_2\} = [(A \ll 16) + D] \cdot B \quad (4.10)$$

where A , B , D are 8-bit inputs, preadder result of $(A \ll 16) + D$ has D in the lower 8 bits, and A starting from the 17th bit onwards, and P_1 and P_2 are the outputs of the two multiplications, with P_2 being the lower 16 bits of P , and P_1 being the rest.

The caveat with DSP packing is that now the accumulation should be realized outside of the DSP. With single INT8 multiplication, 32-bit accumulation can be absorbed inside the DSP logic. However, with packed two INT8 multiplications, 48-bit accumulator precision is not enough to have two 32-bit accumulations. Therefore, accumulations realized outside of the DSP might increase the LUT usage. This trade-off should be carefully considered for the best utilization of the FPGA resources.

Chapter 5

Results

This chapter presents the experimental results and analysis of our integer-only BERT implementation. We evaluate both the training accuracy and inference performance across different hardware platforms.

The chapter is organized as follows. Section 5.1 demonstrates the finetuning process and accuracy results on GLUE benchmark tasks. Section 5.2 evaluates our FPGA accelerator implementation, comparing performance with GPU and Apple platforms, and analyzes resource utilization and design space exploration results.

5.1 Training

We follow the training procedure described in [17] to finetune the I-BERT-base model. We load the pre-trained I-BERT-base model using the transformers library from HuggingFace’s model hub with the model path `kssteven/ibert-roberta-base`. To obtain a model in integer-only precision, we complete the following steps:

1. Finetune from the pre-trained model in full precision on a downstream task and save the checkpoint once the desired accuracy is achieved.
2. Change the model configuration file to enable integer-only quantization mode.
3. Continue finetuning from the saved checkpoint on the same downstream task but now with quantization mode enabled, essentially performing Quantization-Aware Training/Finetuning to recover the accuracy loss due to quantization.

Task	Description
MNLI-m	Determine if a premise entails, contradicts, or is neutral to a hypothesis
QQP	Determine if two questions are semantically equivalent
QNLI	Determine if a question can be answered using a given sentence
SST-2	Classify the sentiment of a sentence as positive or negative
CoLA	Determine if a sentence is grammatically correct
STS-B	Predict the semantic similarity between two sentences
MRPC	Determine if two sentences are paraphrases of each other
RTE	Determine if a hypothesis is entailed by a given premise

Table 5.1: Description of GLUE benchmark tasks

When integer mode is enabled, all the floating-point operations are replaced with their integer-only implementations, and forward pass during training is performed with quantized weights and activations. For backward passes, weights are still updated with floating-point precision. Thus, QAT uses Dynamic Quantization for better convergence. Authors in [17] refer to Yao et al. [39] for more details about the quantization-aware finetuning method for integer-only quantization.

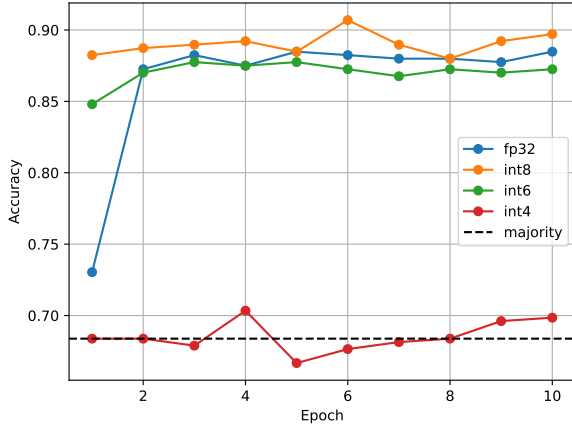
For the training setup, we use an NVIDIA GeForce RTX 4070 SUPER GPU with 12GB of memory. Finetuning results of the I-BERT-base model on the General Language Understanding Evaluation (GLUE [34]) benchmark tasks are reported in Table 5.2. For performance evaluations, we use the standard metrics from the GLUE benchmark. As in [17], the average of two metrics was used for the following tasks: classification accuracy and F1 score for QQP [14] and MRPC [10], Pearson Correlation and Spearman Correlation for STS-B [4], and Matthews Correlation Coefficient for CoLA [36]. Classification accuracy is reported for the remaining tasks: RTE [6], QNLI [28], SST-2 [29], and MNLI-match (MNLI-m) [37]. A description of GLUE benchmark tasks is provided in Table 5.1.

Table 5.2 shows the performance of the I-BERT-base model. The authors in [17] reported that I-BERT in INT8 achieves nearly the same performance as its FP32 counterpart. We reproduce similar results from our finetuning experiments. It is worth noting that the INT8 model can even achieve better performance than the FP32 model on some tasks, such as QQP, SST-2, and MRPC. Figure 5.1 shows that the INT8 models improve their validation accuracy over training epochs, which confirms the effectiveness of QAT.

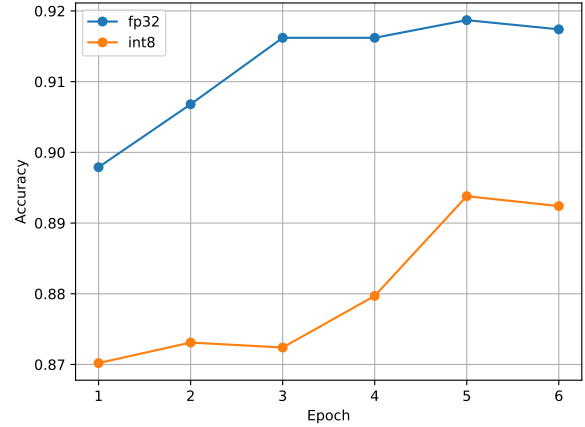
We also report the performance of the I-BERT-base model in lower precision modes, such as INT4 and INT6, on STS-B, MRPC, and RTE tasks. The results are shown in Table 5.3. Although both INT4 and INT6 models achieve lower performance than the INT8 model, the performance drop of INT6 from INT8 is less than 1%.

Model	Precision	MNLI-m	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE
BERT [5]	FP32	84.6	71.2	90.5	93.5	52.1	85.8	88.9	66.4
RoBERTa [6]	FP32	87.6	91.9	92.8	94.8	63.6	91.2	90.2	78.7
I-BERT [7]	FP32	87.8	90.4	92.8	94.6	61.2	91.1	90.9	78.0
I-BERT [7]	INT8	87.5	90.2	92.8	95.2	62.5	90.8	91.1	79.4
I-BERT-f	FP32	86.2	90.1	91.9	94.8	63.2	90.7	90.1	81.2
I-BERT-f	INT8	86.0	90.3	89.4	94.9	62.1	90.0	91.1	79.4

Table 5.2: Performance comparison of different BERT base models on GLUE benchmark tasks (higher is better). I-BERT results are taken from [17], whereas I-BERT-f is the performance result of our finetuning.



(a) MRPC task



(b) QNLI task

Figure 5.1: Validation accuracy of I-BERT-base model during training on MRPC and QNLI tasks

Precision	STS-B	MRPC	RTE
FP32	90.7	90.1	81.2
INT8	90.0	91.1	79.4
INT6	89.0	90.0	78.7
INT4	76.7	75.1	57.8

Table 5.3: Performance comparison of different BERT base models on GLUE benchmark tasks.

Platform	Tech	F (GHz)	DDR (GB)	DDR Bandwidth (GB/s)	L2 (MB)
NVIDIA RTX 4070 SUPER	5nm	1.98-2.48	12	504	48
NVIDIA Jetson Orin Nano	8nm	1.0	8	68	2-4
Apple M1	5nm	3.2	16	70	12
Apple M4	3nm	3.49	16	100	32

Table 5.4: Hardware specifications of evaluated platforms

Specification	Kria KV260	PYNQ-Z1
Device	Zynq UltraScale+ MPSoC	Zynq 7000
Technology	16nm	28nm
CPU Frequency	1.33 GHz	650 MHz
DDR	4GB	0.5GB
LUT	175K	71K
FF	234K	106K
BRAM	144, 648KB	144, 648KB
URAM	64, 2.25MB	-
DSP	1280	220

Table 5.5: FPGA platform specifications

5.2 Inference

We implement our accelerator on two FPGA boards: PYNQ-Z1 and Kria KV260. Additionally, we compare the inference performance across different hardware platforms including NVIDIA RTX 4070 SUPER and Jetson Orin Nano GPUs, and Apple M1 and M4 chips. Tables 5.4 and 5.5 provide the key specifications of these platforms.

From Table 5.6, we can observe that the I-BERT-base model exceeds the on-chip capacity of the FPGA boards. With quantized weights from FP32 to INT8, we reduce the memory footprint of the I-BERT-base model from 501MB to 150MB.

Model	Layers	Embed. Size	Attn. Heads	Parameters	Memory
I-BERT-base	12	768	12	125M	501MB
I-BERT-large	24	1024	16	355M	1.43GB

Table 5.6: Dimensions of I-BERT [17] models in FP32 baseline

Slices	Runtime (ms)
8	6612
2	2562
1	2424

Table 5.7: Impact of number of slices on runtime performance on PYNQ-Z1

5.2.1 Design Space Exploration

We started our design with the goal of implementing pipelined encoder architecture shown in Figure 4.1. At the initial stages of the design, we deployed the slicing approach, discussed in Section 4.4.1, to be able to fit the model on a resource-constrained FPGA on the PYNQ-Z1 board. Runtime results of the slicing approach are shown in Table 5.7: all the systolic arrays are of size 4x4 and the AXI packing factor is 1. We observe that the runtime reduces with decreasing slices, but the reduction is not linear. For example, a decrease from 8 slices to 2 slices results in only 1.93x speedup instead of 4x, and only 1.07x from 2 slices to 1 slice. This is because we hit the computation bottleneck: improving communication speed does not help if we do not increase the systolic array sizes.

We were able to reduce the number of slices to 1, essentially sending the weights only once to the FPGA, and keep all the activations on-chip mostly due to resource optimizations, such as the reuse of one systolic array in the Attention Head module, block-wise output storage, and removal of the separate residual buffers. For example, in the Intermediate stage, we multiply an activation matrix with dimensions of (32, 768) with a weight matrix of dimensions (768, 3072), resulting in an output of (32, 3072). Weights are supplied in blocks of (768,4). Block-wise output storage allows us to store the output of only one block at a time (32,4), instead of the entire matrix of (32,3072), resulting in a significant memory reduction, since the matrix multiplication outputs are in 32-bits.

For our fully on-chip FPGA implementations, we conducted extensive design space exploration as shown in Table 5.8. We vary the systolic array dimensions, AXI packing factor, and operating frequency. On the PYNQ-Z1 board, we achieved a 1.93x speedup by increasing the operating frequency from 50MHz to 100MHz. Further, we optimize through AXI packing to pack 4 elements into a single AXI transaction and double the systolic dimensions for (N_{inter} and N_{layer}) modules that contain the largest matrix multiplications achieving 870ms latency. We can see that the latency is directly proportional to the size of the systolic arrays. However, the largest systolic sizes that can be accommodated are limited by the available FPGA resources, especially on a small FPGA like the PYNQ-Z1. The KV260 board, with larger resources than the PYNQ-Z1, enables larger systolic

Board	F (MHz)	N_{attn}	N_{self}	N_{inter}	N_{layer}	Packing	Runtime (ms)
PYNQ-Z1	50	4	4	4	4	1	3492
PYNQ-Z1	100	4	4	4	4	1	1812
PYNQ-Z1	100	4	4	4	4	4	1764
PYNQ-Z1	100	4	4	8	8	4	870
KV260	100	8	8	8	8	4	446
KV260	100	8	8	8	8	8	441
KV260	100	8	8	16	16	8	235
KV260	100	16	16	16	16	8	132
KV260	100	16	16	16	16	16	131
KV260	200	16	16	16	16	16	66
KV260	250	16	16	16	16	16	53

Table 5.8: Our FPGA inference designs with different frequencies, systolic array dimensions, and AXI packing.

array dimensions. The largest design with 16x16 systolic arrays and AXI packing of 16 achieves 53ms latency at 250MHz. Note that increasing the AXI packing factor gives diminishing returns if we do not increase the systolic size. For example, increasing the AXI packing factor from 4 to 8 on KV260 only reduced the latency by 5ms for the 8x8 systolic array design. This is a compute and communication trade-off that needs to be balanced considering the available AXI bandwidth. We investigated that a packing factor of 16 elements with the available 128-bit interface ($16 \times \text{INT8} = 128\text{-bit}$) is sufficient for the systolic array sizes of 16x16, since we supply one INT8 weight element every cycle to each systolic column.

Table 5.8 also shows that increasing the systolic array dimensions gives consistent improvements in latency, confirming that the matrix multiplications are the main computing components. This is also the reason why we instantiate only single copies of nonlinear function units in a module, e.g., instead of supplying a 4x4 systolic array output to 4 Softmax units in parallel in the Attention Head module, we use a single Softmax unit that accepts a serialized output from the systolic array output memory banks. It can also be observed from the dimensions of the activation matrices vs. the weight matrices, e.g., (32,768) vs. (768,3072). Since the weights are supplied on a stream basis, matrix multiplication takes at least the amount of time it takes to transfer the whole weight matrix. Thus, the time to serially send the activations is negligible compared to the matrix multiplication cycles. Moreover, in our pipelined design, systolic output can be streamed out in overlap while new inputs are loaded.

F (MHz)	100	200	250
Non-Batched design runtime (ms)	131	66.0	53.0
Batched design runtime (ms)	200	100	80.0
Non-Batched design throughput (seq/s)	7.63	15.2	18.9
Batched design throughput (seq/s)	20.0	40.0	50.0
Speedup	2.62	2.64	2.65

Table 5.9: Performance comparison of non-batched and batched designs at different operating frequencies on KV260.

sN	Cycles
2	374
4	335
8	332
16	331

Table 5.10: Attention head execution cycles

Table 5.9 demonstrates the effectiveness of our batch processing implementation with four AXI ports. At 250MHz, while the non-batched design achieves 53ms latency, the batched design processes four sequences in 80ms. Although the total latency increased, we achieved a throughput improvement of 2.6x across different operating frequencies, validating the efficiency of our pipelined architecture for batch processing.

Table 5.8 shows all the main systolic dimensions that perform multiplication with the weights. Two systolic arrays in the attention head multiply activations. Their dimension sN is kept small since their computation is overlapped with the computation of the main systolic arrays. Table 5.10 shows the number of cycles for the attention head execution with different sN . We see that after $sN = 4$, there is almost no improvement since the attention head execution is dominated by the computation of the main systolic array.

5.2.2 Runtime Comparisons

For comparison purposes, we tried to run integer-only inference on the GPU and Apple platforms. In Table 5.11, we show the inference performance comparison for different precision modes. There are three precision modes: full FP32 mode, INT8 weights and activations cast to FP32, and full INT8 mode. We use the source code from HuggingFace’s transformers library which was provided by the authors of I-BERT in [17]. Although, the

Precision	RTX 4070	Apple M1	Apple M4	Jetson Orin Nano
fp32	11.2	15.2	10.4	26.7
fp32(int8)	283	546	417	701
int8	624	511	367	1073

Table 5.11: Impact of quantization on inference latency (ms)

source code provides details of implementing integer-only nonlinear functions, all the data types are still in floating-point precision. Their GitHub page¹ has a note saying that: “Our work is still in progress. Currently, all integer operations are executed with floating-point.”. We modified the source code to remove all the floating-point conversions and use integer-only data types for all the operations for inference purposes. So, full FP32 refer to the original source code and FP32(INT8) is execution with “quant” mode enabled, and INT8 refers to our modified integer-only implementation. We notice that INT8 inference latency is higher than both FP32(INT8) and full FP32. This is because the integer-only implementation is not optimized with the CUDA-specific kernels. Authors of I-BERT mention using TensorRT kernels for INT8 inference that outperform FP32 inference by 2-3x. However, the source kernels were not provided. Furthermore, they only report relative speedup without actual latency measurements.

Therefore, for comparisons we use FP32 implementation, and the results are shown in Table 5.12. For batch size 1, FP32 inference latency ranges from 10.4ms on Apple M4 to 26.7ms on Jetson Orin Nano. Increasing the batch size to 4 improves throughput on all platforms. They all outperform our FPGA implementation mainly due to larger on-chip resources, especially the number of cores, on-chip memory, and higher memory bandwidth. For example, Jetson Orin Nano has 68GB/s memory bandwidth [20], while KV260 has only 16GB/s at 250MHz with four 128-bit AXI ports.

5.2.3 Resource Utilization

The resource utilization for both single and four AXI designs is detailed in Table 5.13. The four AXI design show slightly higher resource usage across all the resources (an increase of at most 8%), except for the DSP utilization which is the same for both designs. We emphasize that Table 5.13 reports the utilization without the DSP packing optimization. After exploration of DSP packing, we observed that implementing accumulation outside of the DSP primitives requires additional LUTs to perform the addition. Thus, we cannot

¹<https://github.com/kssteven418/I-BERT>

Device	Latency (ms)		Throughput (seq/s)	
	Batch 1	Batch 4	Batch 1	Batch 4
RTX 4070	11.2	11.5	89.3	348
Apple M1	15.2	24.8	65.8	161
Apple M4	10.4	16.6	96.2	241
Jetson Orin Nano	26.7	42.7	37.5	93.7
KV260	53.5	80.0	18.9	50
PYNQ-Z1	879	-	1.15	-

Table 5.12: Comparison of inference latency and throughput of different devices

Resource	Available	Single AXI Util (%)	Four AXI Util (%)
LUT	117,120	63,946 (54.6)	68,851 (58.8)
FF	234,240	125,998 (53.8)	135,860 (58.0)
CLB	14,640	14,268 (97.5)	14,575 (99.6)
BRAM	144	97.5 (68.1)	105 (72.9)
URAM	64	41 (100.0)	43 (67.2)
DSP	1,248	1,123 (90.0)	1,123 (90.0)

Table 5.13: FPGA resource utilization for the single and four AXI designs on KV260.

employ DSP packing on KV260 since we are already at the limit of the available CLBs (97.5-99.6%). However, given more LUTs, DSP packing can be used to scale our design to 32x32 systolic arrays by using the same number of DSPs.

Resource utilization breakdown of nonlinear functions on FPGA is shown in Table 5.14. LayerNorm modules consume the most resources, while GELU is relatively lightweight. In total, nonlinear functions account for only 3.56-10.1% of the total resources of the four AXI designs, with the highest proportion being LUT usage at 10.1%.

5.2.4 CPU Offloading

We also include a runtime analysis of nonlinear functions in FP32 precision executed on the ARM CPU of KV260 in Table 5.15. GELU dominates the computation with 37.4ms due to the largest input size, while Softmax and Layer Normalization contribute 2.9ms and 2.3ms, respectively. The total time spent in nonlinear operations amounts to 45.1ms per layer, resulting in 541.2ms across all layers. This means that a design that offloads nonlinear functions to the CPU needs at least 541.2ms to perform nonlinear operations on

Module	LUT	FF	BRAM	DSP
Softmax	1168	1291	0	14
GELU	329	394	0	6
LayerNorm1	2719	1656	1	10
LayerNorm2	2719	1656	1	10
Total Nonlinear	6935	4997	2	40
% of Total Design	10.1	3.68	1.90	3.56

Table 5.14: Resource utilization of nonlinear functions on FPGA.

Operation	Input Dimensions	CPU Runtime (ms)
Softmax	(32, 768)	2.9
LayerNorm1	(32, 768)	2.3
Gelu	(32, 3072)	37.4
LayerNorm2	(32, 768)	2.3
1 Layer	—	44.9
12 Layers	—	538.8

Table 5.15: CPU Offload Operation Runtimes in FP32

a single sequence. This is 10x slower than our FPGA implementation which has a latency of 53.5ms. The reason is that nonlinear function execution on the FPGA is overlapped with the output streaming of the systolic arrays. Therefore, implementing nonlinear functions on an FPGA provides significant speedup compared to offloading to the CPU at the cost of only 10% of additional hardware resources.

5.2.5 Power Consumption

Table 5.16 compares the power consumption and inference latency of our KV260 implementation against Jetson Orin Nano and Apple M4. Our design on the KV260 achieves 53.5ms latency while consuming 12W of power. While the Apple M4 achieves lower latency at 10.4ms, it requires higher power consumption at 30W. The Jetson Orin Nano, operating in two different power modes, demonstrates a trade-off between performance and power consumption - its high-performance mode achieves 26.7ms latency at 11W, while the low-power mode operates at 38.4ms latency using 9W.

Device	Power (W)	Latency (ms)
KV260	12	53.5
Apple M4	30	10.4
Jetson Orin Nano (Low Power)	9	38.4
Jetson Orin Nano (High Performance)	11	26.7

Table 5.16: Power consumption and latency comparison across different devices, batch size of 1.

5.3 Comparison with Prior Works

We compare resource utilization and performance numbers of our design with prior works in Table 5.17 and Table 5.18. We implement the I-BERT-base model which is a finetuned RoBERTa-base model with an integer-only inference. Thus, we can directly compare our design with prior works that also implement the RoBERTa-base model [19, 16, 1]. We have the highest clock frequency of 250MHz among the RoBERTa-base models shown in Table 5.17. It needs to be noted that power measurements of Pushing [20] are reported from the Vivado tool, and not measured during the actual board run as we do. As for the Vivado report, our power consumption for the batched design on KV260 is 5.5W. This puts our design as the most power-efficient implementation among the listed FPGA accelerations. In contrast to encoder-based models, FlightLLM [41] and Pushing [20] deploy decoder-based 7B parameter LLaMA2 model on FPGAs. Pushing [20] shows the highest clock frequency of 300MHz which is mostly due to their low resource utilization shown in Table 5.18. They justify using a dot-product approach with only 128 parallel multipliers, rather than a full matrix multiplication unit, by pointing to the highly vectorized nature of decoder-based models: during the decode phase, only one token is generated at a time. However, we notice that their throughput is only 4.9 tokens/s compared to 55.0 tokens/s of a similar LLaMA2-7B implementation in FlightLLM [41].

We use the least number of resources except for the URAM utilization, after Pushing [20], compared to the other designs. At the same time, we achieve the second highest throughput per DSP of 44.5×10^{-3} , while using significantly fewer BRAMs, LUTs, and FFs compared to NPE [16] which has 66.9×10^{-3} throughput per DSP. We emphasize that given more LUT resources, our design can adopt DSP packing to double the systolic dimensions to reach higher throughput by keeping the same DSP usage. LTrans-OPU [1] reports the highest throughput (222.0 seq/s) among the RoBERTa implementations but requires substantial resources (6,144 DSPs, 1,129 BRAMs).

Most of the works accelerate single-sequence inference, except for FTRANS [19] which

Design	Model	Platform	Freq. (MHz)	Power (W)
FTRANS [19]	RoBERTa-base	Xilinx Virtex VCU118	170	25
NPE [16]	RoBERTa-base	Zynq Z-7100	200	20
LTrans-OPU [1]	RoBERTa-base	Xilinx Alveo U200	200	-
FlightLLM [41]	LLaMA2-7B	Xilinx Alveo U280	225	45
Pushing [20]	LLaMA2-7B	Kria KV260	300	6.6 ¹
Our	RoBERTa-base	Kria KV260	250	12

Table 5.17: Platform and model specifications of different FPGA implementations

Design	Batch Size	DSP	BRAM	URAM	FF	LUT	Lat. (ms)	Thr. (seq/s)	Thr./DSP (10^{-3})
[19]	1	6,531	-	-	507K	451K	10.6	94.3	14.4
[19]	4	6,531	-	-	507K	452K	40.3	99.1	15.2
[19]	8	6,531	-	-	509K	452K	79.0	101	15.5
[19]	16	6,531	-	-	509K	453K	157	102	15.6
[16]	1	2,020	369	0	351K	192K	7.40	135	66.9
[1]	1	6144	1129	0	-	-	4.50	222	36.1
[41]	1	6,345	1,252	792	943K	574K	18.2	55.0 ¹	8.67
[20]	1	291	36.5	10	105K	78K	204	4.9 ¹	16.8
Our	1	1,123	97.5	41	126K	64K	53.5	18.9	16.8
Our	4	1,123	105	43	136K	69K	80.0	50.0	44.5

Table 5.18: Resource usage and performance of different designs: Lat. is Latency (lower is better), Thr. is Throughput (higher is better), Thr./DSP is Throughput per DSP (higher is better). Notes: 1 – decoder based models report tokens per second.

reports performance numbers for batch sizes of 1, 4, 8, and 16. Their resource utilization stays almost on the same level for different batch sizes which is similar to what we observed in our design. However, their latency increases linearly with the increase in batch size, while throughput stays almost the same. This suggests that their design processes each sequence mostly sequentially. In comparison, our design shows a 2.6x throughput improvement for a batch size of 4 compared to single sequence inference, demonstrating the benefits of our pipelined architecture for batch processing. We also highlight that our design is easily scalable given a larger device, and we can anticipate further throughput improvements for larger designs.

We also compare the resource utilization of different nonlinear function implementations in Table 5.19. For all the nonlinear operations, NPE [16] and Pushing [20] report the

Design	Unit	Data Format	LUT	FF	DSP	URAM	BRAM
NPE [16]	Vector	16-bit fixed	38K	13K	32	0	32
Pushing [20]	Vector	FP16	29K	40K	24	3	6.5
Our	Vector	INT32	6.9K	5K	40	0	2
Basic-Split [31]	Softmax	32-bit fixed	18K	16K	0	0	0
Booth Multiplier [5]	Softmax	16-bit fixed	2K	1.4K	0	0	0
Our	Softmax	INT32	1.1K	1.2K	14	0	0

Table 5.19: Resource utilization comparison of different nonlinear function implementations

resource utilization of their vector units that implement all the nonlinear functions. Thus we also report the total resource usage of all of our nonlinear units combined. NPE [16] implements piecewise polynomial functions using 16-bit fixed-point arithmetic, while Pushing [20] adopts FP16 floating-point arithmetic. Our design uses INT32 arithmetic, achieving significantly lower resource utilization in terms of LUTs, FFs, and BRAMs. We use slightly more DSPs due to second-order polynomial functions used for nonlinear approximations.

For softmax implementations, the Basic-Split approach [31] decomposes exponential calculations into LUT-based operations using 32-bit fixed-point arithmetic, requiring 18K LUTs and 16K FFs. Chen et al. [5] design a hybrid exponent unit with 16-bit fixed-point precision and dedicated Booth multiplier to reduce resource usage to 2K LUTs and 1.4K FFs. Our INT32 softmax implementation further reduces LUT usage to 1.1K and FFs to 1.2K while utilizing 14 DSPs.

Chapter 6

Conclusion

6.1 Conclusion

This thesis has investigated accelerating BERT model inference on FPGAs through integer-only computation. The key contributions of this thesis are:

- Implementation of integer-only nonlinear functions (GELU, Softmax, Layer Normalization) on FPGAs that utilize only 10% of the total design resources. Our on-chip implementation is at least 10x faster than the CPU offloading approach, achieving 53ms latency at 250MHz for a single sequence on the KV260 board.
- Optimizations to solve on-chip resource limitations, enabling us to keep all the activations on-chip. We explored different BRAM/URAM configurations for efficient synthesis, merged double buffers, and implemented block-wise output storage. We adopted an AXI packing technique to send multiple 8-bit weights in wider transactions (up to 128-bit), which allowed faster weight transfer. We applied a resource-sharing strategy to reuse a single systolic array in the attention head module to free resources for larger systolic dimensions.
- A novel batched, pipelined architecture with four parallel processing stages (Attention Head, Self Attention Output, Intermediate, and Layer Output). We analyzed potential pipeline stall cases and placed pipeline buffers to eliminate module idle times and prevent deadlocks. We achieved a consistent 2.6x speedup in throughput over a non-batched design on a wide range of operating frequencies.

These optimizations collectively enabled us to implement a fully on-chip design with 16x16 systolic arrays operating at 250MHz on KV260, achieving 53ms latency for single-sequence inference and processing four sequences in parallel with 80ms total latency. The experimental results show that our batched design outperforms the non-batched approach by processing four sequences simultaneously. Our implementation demonstrates that by removing floating-point operations and using efficient integer approximations as shown in [17], combined with our optimization techniques, it is possible to achieve a high throughput on FPGAs while maintaining model accuracy.

6.2 Limitations and Future Work

Despite the above results, several limitations and opportunities for future work remains:

- **Larger models:** At present, we have used I-BERT-base model for the analysis, but integer-only [17] approach can be also applied to larger transformer models.
- **Decoder-based models:** This work has been performed on the encoder-only model (BERT). Implementing decoder-based models will enable us to extend the application of our techniques to generative models.
- **Lower precision quantization:** We have also applied successful QAT to reduce the precision of the model from FP32 to INT8 and INT6. Further work on even lower precisions, such as INT4 and INT3, may lead to more memory savings. Our initial experiments show that INT4 causes severe accuracy drop, thus the research aim is to address this issue.
- **Hardware scaling:** Implementation on larger FPGA boards and devices. Our design is easily scalable when more on-chip resources are available.
- **Comparison with floating-point implementations:** A direct comparison with FPGA implementations that use floating-point would have given a better understanding of the advantages of using only integers in terms of resources, time, and power consumption.

References

- [1] Yueyin Bai, Hao Zhou, Keqing Zhao, Manting Zhang, Jianli Chen, Jun Yu, and Kun Wang. LTrans-OPU: A Low-Latency FPGA-Based Overlay Processor for Transformer Networks. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 283–287, 2023.
- [2] Andrea Belano, Yvan Tortorella, Angelo Garofalo, Luca Benini, Davide Rossi, and Francesco Conti. A Flexible Template for Edge Generative AI with High-Accuracy Accelerated SoftMax & GELU, 2024.
- [3] Aishwarya Bhandare, Vamsi Sripathi, Deepthi Karkada, Vivek Menon, Sun Choi, Kushal Datta, and Vikram A. Saletore. Efficient 8-Bit Quantization of Transformer Neural Machine Language Translation Model. *CoRR*, abs/1906.00532, 2019.
- [4] Daniel Cer, Mona Diab, Eneko Agirre, Inigo Lopez-Gazpio, and Lucia Specia. SemEval-2017 Task 1: Semantic Textual Similarity Multilingual and Crosslingual Focused Evaluation. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*. Association for Computational Linguistics, 2017.
- [5] Ke Chen, Yue Gao, Haroon Waris, Weiqiang Liu, and Fabrizio Lombardi. Approximate Softmax Functions for Energy-Efficient Deep Neural Networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 31(1):4–16, 2023.
- [6] Ido Dagan, Oren Glickman, and Bernardo Magnini. The PASCAL Recognising Textual Entailment Challenge. In *Proceedings of the First International Conference on Machine Learning Challenges: Evaluating Predictive Uncertainty Visual Object Classification, and Recognising Textual Entailment*, MLCW’05, page 177–190, Berlin, Heidelberg, 2005. Springer-Verlag.
- [7] William Dally. High-Performance Hardware for Machine Learning. Tutorial presented at the 29th Conference on Neural Information Processing Systems (NIPS), 2015.

- [8] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. GPT3.int8(): 8-bit Matrix Multiplication for Transformers at Scale. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 30318–30332. Curran Associates, Inc., 2022.
- [9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.
- [10] William B. Dolan and Chris Brockett. Automatically Constructing a Corpus of Sentential Paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*, 2005.
- [11] Gaoming Du, Chao Tian, Zhenmin Li, Duoli Zhang, Yongsheng Yin, and Yiming Ouyang. Efficient Softmax Hardware Architecture for Deep Neural Networks. In *Proceedings of the 2019 Great Lakes Symposium on VLSI, GLSVLSI '19*, page 75–80, New York, NY, USA, 2019. Association for Computing Machinery.
- [12] Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig. Deep Learning with INT8 Optimization on Xilinx Devices. <https://docs.amd.com/v/u/en-US/wp486-deep-learning-int8>, April 2017. White Paper: UltraScale and UltraScale+ FPGAs, WP486 (v1.0.1).
- [13] Yu Gao, Juan Camilo Vega, and Paul Chow. The Feasibility of Implementing Large-Scale Transformers on Multi-FPGA Platforms, 2024.
- [14] Shankar Iyer, Nikhil Dandekar, and Kornél Csernai. First Quora Dataset Release: Question Pairs, 2017.
- [15] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.
- [16] Hamza Khan, Asma Khan, Zainab Khan, Lun Bin Huang, Kun Wang, and Lei He. NPE: An FPGA-based Overlay Processor for Natural Language Processing. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*,

- FPGA '21, page 227, New York, NY, USA, 2021. Association for Computing Machinery.
- [17] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. I-BERT: Integer-only BERT Quantization. In Marina Meila and Tong Zhang, editors, *38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 5506–5518. PMLR, 18–24 Jul 2021.
 - [18] Liangzhen Lai, Naveen Suda, and Vikas Chandra. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. *ArXiv*, abs/1801.06601, 2018.
 - [19] Bingbing Li, Santosh Pandey, Haowen Fang, Yanjun Lyv, Ji Li, Jieyang Chen, Mimi Xie, Lipeng Wan, Hang Liu, and Caiwen 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.
 - [20] Jindong Li, Tenglong Li, Guobin Shen, Dongcheng Zhao, Qian Zhang, and Yi Zeng. Pushing up to the Limit of Memory Bandwidth and Capacity Utilization for Efficient LLM Decoding on Embedded FPGA, 2025.
 - [21] Jinhao Li, Jiaming Xu, Shan Huang, Yonghua Chen, Wen Li, Jun Liu, Yaoxiu Lian, Jiayi Pan, Li Ding, Hao Zhou, Yu Wang, and Guohao Dai. Large Language Model Inference Acceleration: A Comprehensive Hardware Perspective, 2025.
 - [22] Zhengang Li, Mengshu Sun, Alec Lu, Haoyu Ma, Geng Yuan, Yanyue Xie, Hao Tang, Yanyu Li, Miriam Leeser, Zhangyang Wang, Xue Lin, and Zhenman Fang. Auto-ViT-Acc: An FPGA-Aware Automatic Acceleration Framework for Vision Transformer with Mixed-Scheme Quantization. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 109–116, 2022.
 - [23] Zhikai Li and Qingyi Gu. I-ViT: Integer-only Quantization for Efficient Vision Transformer Inference, 2023.
 - [24] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR*, abs/1907.11692, 2019.
 - [25] Alberto Marchisio, Davide Dura, Maurizio Capra, Maurizio Martina, Guido Masera, and Muhammad Shafique. SwiftTron: An Efficient Hardware Accelerator for Quantized Transformers, 2023.

- [26] Kyle Marino, Pengmiao Zhang, and Viktor K. Prasanna. ME- ViT: A Single-Load Memory-Efficient FPGA Accelerator for Vision Transformers . In *2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 213–223, Los Alamitos, CA, USA, December 2023. IEEE Computer Society.
- [27] Nicholas Marino et al. An FPGA Accelerator for BERT. *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021.
- [28] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In Jian Su, Kevin Duh, and Xavier Carreras, editors, *2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics.
- [29] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In David Yarowsky, Timothy Baldwin, Anna Korhonen, Karen Livescu, and Steven Bethard, editors, *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA, October 2013. Association for Computational Linguistics.
- [30] Mengshu Sun, Haoyu Ma, Guoliang Kang, Yifan Jiang, Tianlong Chen, Xiaolong Ma, Zhangyang Wang, and Yanzhi Wang. VAQF: Fully Automatic Software-Hardware Co-Design Framework for Low-Bit Vision Transformer, 2022.
- [31] Qiwei Sun, Zhixiong Di, Zhengyang Lv, Fengli Song, Qianyin Xiang, Quanyuan Feng, Yibo Fan, Xulin Yu, and Wenqiang Wang. A High Speed SoftMax VLSI Architecture Based on Basic-Split. In *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pages 1–3, 2018.
- [32] Qiwei Sun, Zhixiong Di, Zhengyang Lv, Fengli Song, Qianyin Xiang, Quanyuan Feng, Yibo Fan, Xulin Yu, and Wenqiang Wang. A High Speed SoftMax VLSI Architecture Based on Basic-Split. In *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pages 1–3, 2018.
- [33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *Advances in Neural Information Processing Systems*, 2017.

- [34] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding, 2019.
- [35] Teng Wang, Lei Gong, Chao Wang, Yang Yang, Yingxue Gao, Xuehai Zhou, and Huaping Chen. ViA: A Novel Vision-Transformer Accelerator Based on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4088–4099, 2022.
- [36] Alex Warstadt, Amanpreet Singh, and Samuel R. Bowman. Neural Network Acceptability Judgments. *Transactions of the Association for Computational Linguistics*, 7:625–641, 2019.
- [37] Adina Williams, Nikita Nangia, and Samuel Bowman. A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference. In Marilyn Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.
- [38] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.
- [39] Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael W. Mahoney, and Kurt Keutzer. HAWQ-V3: Dyadic Neural Network Quantization. *arXiv preprint arXiv:2011.10680*, 2020.
- [40] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8BERT: Quantized 8Bit BERT. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*, page 36–39. IEEE, December 2019.
- [41] Shulin Zeng, Jun Liu, Guohao Dai, Xinhao Yang, Tianyu Fu, Hongyi Wang, Wenheng Ma, Hanbo Sun, Shiyao Li, Zixiao Huang, Yadong Dai, Jintao Li, Zehao Wang, Ruoyu Zhang, Kairui Wen, Xuefei Ning, and Yu Wang. FlightLLM: Efficient Large Language Model Inference with a Complete Mapping Flow on FPGAs. In *Proceedings of the 2024*

ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '24, page 223–234, New York, NY, USA, 2024. Association for Computing Machinery.

- [42] Xiandong Zhao, Ying Wang, Xuyi Cai, Cheng Liu, and Lei Zhang. Linear Symmetric Quantization of Neural Networks for Low-Precision Integer Hardware. In *International Conference on Learning Representations*, 2020.