

# Acceleration of Integer Transformer Models Via Structured Resource Management Using FPGAs

by

Omar Elayat

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

© Omar Elayat 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

The widespread adoption of [Large Language Models \(LLMs\)](#) in various applications has pushed the demand for efficient hardware acceleration beyond the capabilities of traditional platforms. Due to their highly parallel architecture and ease of deployment, [Field-Programmable Gate Arrays \(FPGAs\)](#) are widely used to accelerate [LLMs](#). However, the [FPGAs](#)' limited on-chip memory resources are still too limited to accommodate the trained models. While existing [FPGA](#)-based solutions have demonstrated promising throughput and energy efficiency, they often rely on abundant fabric resources, assume high-bandwidth devices that are not suitable for deployment at the edge, or employ highly customized acceleration architectures that are not scalable with the advancements of the [LLMs](#) architectures. This thesis addresses these challenges by proposing a novel on-chip resources manager architecture for integer encoder-based transformer inference, with a focus on [Bidirectional Encoder Representations from Transformers \(BERT\)](#) models. We target resource-constrained [FPGAs](#) with limited memory bandwidth. We show that, through structured operation scheduling and resource-sharing, significant performance improvements can be achieved. The proposed resource-shared infrastructure is also designed to be modular, allowing newly introduced computation blocks to be easily integrated into the accelerator, without requiring major modifications or incurring additional off-chip data movement.

Demonstrated on a fully quantized integer-only variant of the [BERT](#) model as a representative workload, the proposed system achieves 2.32x latency improvement over the baseline custom accelerator, 1.17x over Jetson Orin Nano GPU, and at least 23.63x over CPU. The design is validated on two [FPGAs](#): the PYNQ-Z1 as a low-end proof-of-concept and the KV260 as a mid-range deployment target.

## Acknowledgements

I would like to thank my supervisor, Prof. Vincent Gaudet, for his unwavering support, kindness, and encouragement from the very first day of this journey. His continued support and motivation played a pivotal role in helping me navigate both the challenges and the successes of this thesis. I would also like to thank my co-supervisor, Prof. Mohamed Elmasry, for his guidance and support.

I am grateful to Prof. Nachiket Kapre for the opportunity to collaborate with his research group, the Waterloo Configurable Architectures group, and for his valuable feedback throughout this work. I am especially thankful to Marat Bekmyrza, Jehanzeb Mirza, and Emir Geuvara for their thoughtful discussions. Their collaboration helped me shape the outcome of this thesis.

# Table of Contents

|                                                 |             |
|-------------------------------------------------|-------------|
| <b>Author’s Declaration</b>                     | <b>ii</b>   |
| <b>Abstract</b>                                 | <b>iii</b>  |
| <b>Acknowledgements</b>                         | <b>iv</b>   |
| <b>List of Figures</b>                          | <b>ix</b>   |
| <b>List of Tables</b>                           | <b>xi</b>   |
| <b>List of Abbreviations</b>                    | <b>xiii</b> |
| <b>1 Introduction</b>                           | <b>1</b>    |
| 1.1 Thesis Contributions . . . . .              | 1           |
| 1.2 Thesis Organization . . . . .               | 2           |
| <b>2 Literature Review</b>                      | <b>3</b>    |
| 2.1 Transformer-based Language Models . . . . . | 3           |

|          |                                                            |           |
|----------|------------------------------------------------------------|-----------|
| 2.1.1    | Transformer Architecture Overview . . . . .                | 4         |
| 2.2      | Hardware Accelerators for Language Models . . . . .        | 5         |
| 2.2.1    | FPGA-Based Accelerators . . . . .                          | 5         |
| 2.2.2    | IBERT Accelerator Reference Baseline . . . . .             | 7         |
| 2.2.3    | General-Purpose Accelerators on CPUs and GPUs . . . . .    | 10        |
| 2.2.4    | ASIC-Based Accelerators . . . . .                          | 11        |
| 2.2.5    | In-Memory Accelerators . . . . .                           | 11        |
| 2.3      | Algorithmic Methods for Efficient Inference . . . . .      | 13        |
| 2.4      | On-Chip Memory Management . . . . .                        | 15        |
| <b>3</b> | <b>Target Model and Deployment Strategy</b>                | <b>18</b> |
| 3.1      | Target Model and Hardware . . . . .                        | 18        |
| 3.1.1    | IBERT Model Overview . . . . .                             | 19        |
| 3.1.2    | Model Deployment . . . . .                                 | 20        |
| 3.1.3    | FPGA Platform Selection . . . . .                          | 23        |
| 3.2      | Platforms Constraints, and Acceleration Strategy . . . . . | 25        |
| <b>4</b> | <b>Hardware Architecture</b>                               | <b>28</b> |
| 4.1      | Systolic Matrix-Matrix Multiplication . . . . .            | 28        |
| 4.2      | Resource Scheduling . . . . .                              | 31        |
| 4.2.1    | Phase Scheduler and Control Synchronization . . . . .      | 33        |

|          |                                                                |           |
|----------|----------------------------------------------------------------|-----------|
| 4.2.2    | Operands Routing and Runtime Dimension Configuration . . . . . | 38        |
| 4.2.3    | Memory Bank Logic and Access Control . . . . .                 | 40        |
| 4.2.4    | Hazard-Resilient Double Buffering Controller . . . . .         | 43        |
| 4.2.5    | Address Generation Algorithms . . . . .                        | 46        |
| 4.2.6    | Multi-AXI Stream Support . . . . .                             | 50        |
| 4.2.7    | Streaming and Management of NL Vectors . . . . .               | 52        |
| 4.2.8    | Auxiliary Functions . . . . .                                  | 53        |
| 4.3      | Requantization Unit . . . . .                                  | 54        |
| 4.4      | Non-Linear Unit and Matrix-Addition . . . . .                  | 54        |
| <b>5</b> | <b>Experimental Setup and Results</b>                          | <b>56</b> |
| 5.1      | Experimental settings . . . . .                                | 56        |
| 5.2      | Performance Evaluation . . . . .                               | 57        |
| 5.2.1    | Bandwidth Constraints in Matrix Operations . . . . .           | 60        |
| 5.3      | Resource Consumption Analysis . . . . .                        | 62        |
| 5.4      | Power Analysis and Comparison on Different Platforms . . . . . | 63        |
| 5.5      | Comparison with Existing FPGA Research . . . . .               | 64        |
| 5.5.1    | Comparison with IBERT Baseline . . . . .                       | 66        |
| <b>6</b> | <b>Conclusion</b>                                              | <b>68</b> |
| 6.1      | Conclusion . . . . .                                           | 68        |
| 6.2      | Limitations and Future Work . . . . .                          | 69        |



# List of Figures

|     |                                                                                       |    |
|-----|---------------------------------------------------------------------------------------|----|
| 2.1 | Hardware architecture of the baseline accelerator . . . . .                           | 8  |
| 3.1 | Flow of IBERT Inference. . . . .                                                      | 21 |
| 3.2 | Self-Attention block diagram. . . . .                                                 | 22 |
| 3.3 | Self-out sublayer block diagram. . . . .                                              | 22 |
| 3.4 | Intermediate sublayer block diagram. . . . .                                          | 23 |
| 3.5 | Layer-out sublayer block diagram. . . . .                                             | 23 |
| 3.6 | IBERT inference on embedded platforms . . . . .                                       | 26 |
| 4.1 | Phased execution of a $2 \times 2$ systolic multiplier with double buffering. . . . . | 30 |
| 4.2 | Hardware architecture of the accelerator . . . . .                                    | 31 |
| 4.3 | Memory manager top module. . . . .                                                    | 32 |
| 4.4 | IBERT self-attention sub-layer schedule . . . . .                                     | 33 |
| 4.5 | IBERT intermediate sub-layer schedule . . . . .                                       | 37 |
| 4.6 | FSM for ping-pong memory control . . . . .                                            | 45 |
| 4.7 | Illustration of AXI-Aligned Streaming layout . . . . .                                | 46 |

|      |                                                                              |    |
|------|------------------------------------------------------------------------------|----|
| 4.8  | Double buffering write scheme . . . . .                                      | 47 |
| 4.9  | Systolic Output Writeback . . . . .                                          | 48 |
| 4.10 | AXI-compatible streaming . . . . .                                           | 49 |
| 4.11 | Multi-AXI memory manager architecture . . . . .                              | 51 |
| 5.1  | Inference runtime improvement factor relative to lowest-bandwidth (0.1 GB/s) | 58 |
| 5.2  | Percentage of matrix-matrix multiplication time . . . . .                    | 61 |

# List of Tables

|    |                                                                                                               |    |
|----|---------------------------------------------------------------------------------------------------------------|----|
| 1  | Architectural comparison of common transformer-based models. . . . .                                          | 4  |
| 2  | FPGA-based language models inference accelerator . . . . .                                                    | 10 |
| 3  | Extended breakdown of LLMs-Transformer accelerators. . . . .                                                  | 13 |
| 4  | IBERT GLUE scores . . . . .                                                                                   | 19 |
| 5  | Comparison of resources between PYNQ-Z1 and KV260 boards. . . . .                                             | 24 |
| 6  | Memory Layouts and Offsets for the Memory Manager . . . . .                                                   | 40 |
| 7  | Memory Group Access Behavior and Logic Mapping . . . . .                                                      | 41 |
| 8  | Read/Write Behavior Interpretation . . . . .                                                                  | 42 |
| 9  | Double-buffer controller output assignments. . . . .                                                          | 44 |
| 10 | Memory Layout for the stream-to-vector Unit. . . . .                                                          | 53 |
| 11 | Measured runtime and software overhead for different configurations. . . .                                    | 57 |
| 12 | Latency, throughput, and DSP-normalized efficiency for the top-performing<br>systolic configurations. . . . . | 59 |

|    |                                                                                                         |    |
|----|---------------------------------------------------------------------------------------------------------|----|
| 13 | Performance and bandwidth utilization under different DMA bandwidths<br>after the elbow points. . . . . | 60 |
| 14 | Dimensions of I-BERT [31] models at sequence length of 32. . . . .                                      | 61 |
| 15 | FPGA resource utilization for elbow points configurations. . . . .                                      | 63 |
| 16 | Resource breakdown per module for the 32×32 configuration at 19.2 GB/s                                  | 63 |
| 17 | Hardware specifications of the evaluated GPUs. . . . .                                                  | 64 |
| 18 | Power consumption and latency comparison with embedded GPUs. . . . .                                    | 64 |
| 19 | Performance comparison across FPGA-based LLM frameworks. . . . .                                        | 65 |

# List of Abbreviations

**ATTN** IBertAttention sub-layer [21](#), [34](#)

**AXI** advanced extensible interface [2](#), [12](#), [16](#), [17](#), [24](#), [33](#), [36](#), [38–40](#), [42](#), [43](#), [50–52](#), [57](#), [58](#), [63](#), [68](#), [70](#)

**BERT** Bidirectional Encoder Representations from Transformers [iii](#), [5–8](#), [10](#), [14](#), [18–20](#)

**BRAM** block RAM [7](#), [9](#), [16](#), [17](#), [20](#), [24–26](#), [29](#), [32](#), [34](#), [36](#), [38–41](#), [47](#), [62](#), [63](#), [66](#), [69](#)

**CAM** content addressable memory [11](#)

**CMOS** complementary metal–oxide–semiconductor [12](#)

**CNN** Convolutional Neural Network [5](#), [6](#), [14](#)

**DDR** double data rate [6](#), [24](#), [25](#), [32](#), [50](#), [52](#), [57](#), [59](#)

**DMA** Direct Memory Access [2](#), [9](#), [25](#), [26](#), [41](#), [57–63](#), [66](#), [68](#), [70](#)

**DRAMs** dynamic random access memories [7](#), [12](#)

**DSP** digital signal processing slice [2](#), [6](#), [7](#), [16](#), [20](#), [23](#), [24](#), [26](#), [32](#), [59](#), [60](#), [62](#), [64](#), [66](#), [69](#)

**FFN** feed-forward networks [4](#), [10](#), [16](#), [20](#)

**FP32** 32-bit floating point [5](#), [14](#), [19](#)

**FPGA** Field-Programmable Gate Array [iii](#), [1](#), [2](#), [5–9](#), [11](#), [12](#), [15–20](#), [23–26](#), [31](#), [32](#), [56](#), [64](#),  
[66](#), [68](#), [69](#)

**FSM** finite state machine [17](#), [26](#), [30](#), [33–37](#), [39](#), [41](#), [43](#), [44](#), [46](#), [54](#)

**GELU** gaussian error linear unit [14](#), [19](#), [31](#), [34](#), [38](#), [62](#)

**GLUE** General Language Understanding Evaluation [19](#)

**HBM** high bandwidth memory [6](#), [15](#), [64](#)

**INT32** 32-bit integer [14](#), [19](#), [22](#), [23](#), [54](#)

**INT8** 8-bit integer [14](#), [19](#), [22](#), [23](#), [54](#)

**INTER** IBERTIntermediate sub-layer [21](#)

**LayerNorm** layer normalization [14](#), [16](#), [19](#), [20](#), [31](#), [34](#), [62](#)

**LLM** Large Language Model [6](#), [7](#), [11](#), [15](#), [25](#), [53](#), [69](#)

**LLMs** Large Language Models [iii](#), [1](#), [11](#)

**LM** language model [4](#), [9](#), [20](#), [28](#)

**LUT** look-up table [6](#), [7](#), [9](#), [16](#), [26](#), [32](#), [36](#), [62](#), [63](#)

**MAC** multiply and accumulate [28](#)

**MATADD** matrix addition unit [28](#), [54](#)

**MLM** masked language modeling [3](#)

**MM** matrix-matrix [1](#), [15–17](#), [28](#), [29](#), [31](#), [33](#), [36–38](#), [41](#), [50](#), [54](#), [60–62](#)

**MM2S** memory-mapped to stream [39](#), [50](#), [51](#)

**MNGR** on-chip memory manager [1](#), [17](#), [28](#), [30–34](#), [36](#), [40](#), [42](#), [46](#), [50](#), [52](#), [54](#), [55](#), [62](#), [66](#), [68](#), [70](#)

**NL** non-linear [5](#), [6](#), [10](#), [14](#), [16](#), [19](#), [20](#), [25](#), [33](#), [34](#), [38](#), [39](#), [52](#), [54](#), [70](#)

**NLP** natural language processing [3](#)

**NLU** non-linear processing unit [28](#), [31](#), [54](#), [62](#)

**NSP** next sentence prediction [3](#), [4](#)

**NVM** non-volatile memory [12](#)

**OUTPUT** IBERTOutput sub-layer [21](#)

**PE** processing element [7](#), [28](#)

**PL** programmable logic [15](#), [20](#), [42](#), [50](#)

**PS** processing system [15](#), [50](#)

**PTQ** post-training quantization [13](#)

**QAT** quantization-aware training [14](#)

**RAMs** random access memories [16](#)

**ReLU** rectified linear unit [14](#), [54](#)

**ReRAMs** resistive random access memories [11](#)

**RU** requantization unit [28](#), [31](#), [54](#)

**S2V** stream-to-vector [32](#), [34](#), [36](#), [38](#), [52](#), [54](#)

**SELFOUT** IBertSelfOutput sub-layer [21](#)

**Softmax** softmax function [10](#), [11](#), [14](#), [16](#), [19](#), [31](#), [34–36](#)

**URAM** Xilinx ultra RAM [9](#), [17](#), [25](#), [26](#), [36](#), [41](#), [47](#), [62](#)

# Chapter 1

## Introduction

The growing adoption of [LLMs](#) in edge applications — such as offline voice assistants, document summarization on mobile devices, and real-time translation — has exposed the limitations of conventional platforms in meeting the computational and memory demands of transformer inference. While [FPGAs](#) offer a promising balance of low latency and energy efficiency, most existing accelerator designs assume high-end devices with abundant resources and memory bandwidth [59, 23, 24]. Deploying LLMs on edge-class [FPGAs](#) introduces challenges in model fitting, bandwidth management, and scheduling efficiency.

### 1.1 Thesis Contributions

This thesis targets the challenges of model fitting and memory reuse by designing a modular resource-managed accelerator for integer encoder transformer inference on edge-class [FPGAs](#). The system is built around a double-buffered systolic [matrix-matrix \(MM\)](#) multiplication engine, controlled by a structured [on-chip memory manager \(MNGR\)](#) that sched-

ules operand routing, and minimizes off-chip transfers. The main contributions of this thesis are:

- A double-buffered centralized on-chip memory manager that supports multi-advanced extensible interface (AXI) stream configurations without relying on back-end Direct Memory Accesss (DMA) multiplexing and synchronization, achieving up to 82.9% bandwidth utilization with minimal resources overhead.
- A resource-managed FPGA accelerator for integer-only encoder-based transformers inference. It enables efficient operand management across the accelerator blocks based on a configured dataflow schedule of an integer-only encoder transformer.
- Demonstration of the full accelerator on PYNQ-Z1 and KV260 platforms with IB-ERT [31] as a test case, achieving  $2.32\times$  and  $2.14\times$  latency and energy efficiency improvements over baseline, respectively. Moreover, the resource-managed design demonstrated comparable throughput-per-digital signal processing slice (DSP) to the state-of-the-art FPGA-based designs.

## 1.2 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 reviews related work on transformer acceleration, quantization methods, and memory scheduling. Chapter 3 breaks down the model chosen for evaluation and outlines deployment strategy on edge-class FPGAs. Chapter 4 details the hardware architecture. Chapter 5 presents the experimental setup, performance metrics, and comparison with existing frameworks. Finally, Chapter 6 concludes the thesis and discusses limitations and directions for future work.

# Chapter 2

## Literature Review

### 2.1 Transformer-based Language Models

Transformer-based language models are a class of deep learning models that use the transformer architecture [53] as a backbone for modeling language. They are trained on large-scale text corpora to predict, generate, or classify sequences of tokens. These models are pretrained on unsupervised objectives and then fine-tuned for specific [natural language processing \(NLP\)](#) tasks such as question answering, summarization, classification, and translation.

Prominent examples include BERT [14], RoBERTa [39], GPT [48], and LLaMA [51] have achieved unprecedented quality of results in a wide range of [NLP](#) tasks. GPT [48] and LLaMA [51] adopt decoder-style architectures optimized for generative tasks such as open-ended question answering, summarization, and instruction following. In contrast, BERT [14] introduced an encoder-only architecture trained using [masked language modeling \(MLM\)](#) and [next sentence prediction \(NSP\)](#), which is widely used for classification,

question answering, and named entity recognition. RoBERTa [39] builds upon BERT [14] by removing the NSP objective, applying dynamic masking, and training on larger corpora with increased batch sizes. While BERT [14] and RoBERTa [39] are architecturally identical, they differ in their pretraining methodologies. Pretraining refers to the initial training phase of a language model (LM) on large unlabeled text corpora using self-supervised objectives, enabling the model to learn general language representations before task-specific fine-tuning.

### 2.1.1 Transformer Architecture Overview

Transformer-based LMs are defined by a set of core architectural parameters: Layers refer to the number of transformer blocks stacked sequentially; Embedding size indicates the dimensionality of token and position embeddings; Attention heads denote the number of parallel attention mechanisms used per layer; and feed-forward networks (FFN) hidden size specifies the dimension of the intermediate FFN within each layer.

| Model             | Layers | Embed. Size | Attn. Heads | FFN Hidden Size | Parameters |
|-------------------|--------|-------------|-------------|-----------------|------------|
| I-BERT-base [31]  | 12     | 768         | 12          | 3072            | 125M       |
| BERT-base [14]    | 12     | 768         | 12          | 3072            | 110M       |
| RoBERTa-base [39] | 12     | 768         | 12          | 3072            | 125M       |
| GPT-2 small [48]  | 12     | 768         | 12          | 3072            | 117M       |
| LLaMA-7B [51]     | 32     | 4096        | 32          | 11008           | 7B         |

Table 1: Architectural comparison of common transformer-based models.

Despite sharing the same architecture, models can differ in total parameter count due to variations in vocabulary size, whether embedding weights are shared with output layers, and the presence or absence of auxiliary layers. For example, RoBERTa and IBERT use larger vocabularies and untied weights, resulting in higher parameter counts than BERT-

base, even though their core architectures are identical.

IBERT [31], the evaluation model used in this thesis, builds upon RoBERTa’s [39] architecture but replaces all 32-bit floating point (FP32) operations with integer-only arithmetic, including custom approximations for non-linear (NL) functions. This enables highly efficient inference with quantized weights, activations, and computations. Architecturally, IBERT remains equivalent to BERT [14] and RoBERTa [39], making it suitable for fair comparison with prior BERT-based FPGAs accelerators. The key distinction lies in its quantization strategy and hardware-friendly execution model, not in its structural design. Table 1 shows an architectural comparison of common transformer-based models. A detailed breakdown of the IBERT model and its hardware implications is provided in Chapter 3.

## 2.2 Hardware Accelerators for Language Models

### 2.2.1 FPGA-Based Accelerators

Recent advances in FPGA-based accelerators have significantly influenced the design of deep neural network inference systems. Early work on Convolutional Neural Network (CNN) accelerators demonstrated that even resource-constrained FPGAs could deliver substantial performance gains through careful design and resource optimization [19, 10]. For instance, Angel Eye [19] provided a comprehensive design flow for mapping CNN onto embedded FPGAs, demonstrating that careful partitioning and operation scheduling could yield efficient hardware implementations. Similarly, Chen et al. [10] proposed a CNN accelerator that balanced computational density with available memory resources.

As deep learning evolved toward transformer architectures, the demand for efficient

hardware acceleration became even more critical. The transformer architecture introduced by Vaswani et al. [53] incorporates multi-head self-attention and deep feed-forward networks, forming the basis for models like BERT [14], RoBERTa [39], and LLaMA 2 [40], which often contain hundreds of millions to billions of parameters. This exponential increase in model complexity and memory demands compared to CNN has driven the development of specialized hardware accelerators.

Researchers have developed FPGA-based architectures tailored to transformers’ computational patterns. Cloud-class solutions such as DFX [23] and FlightLLM [59] were among the first to demonstrate efficient Large Language Model (LLM) inference on high-end devices like the Xilinx Alveo U280, leveraging high bandwidth memory (HBM) and dot-engine architectures to support decoder-style workloads.

Other approaches have explored fully customized architectures [59, 33, 17, 20, 21, 28, 42, 54]. FTRANS [33] presents a heterogeneous design that offloads embedding lookups to external double data rate (DDR) memory while mapping all attention and feed-forward operations on-chip. It performs hardware-software co-design, where model compression is performed offline, and matrix multiplications and NL operations are computed on the FPGA. FTRANS achieves up to  $27\times$  speedup and  $81\times$  energy efficiency over a CPU baseline and  $8.8\times$  energy efficiency over an RTX5000 baseline GPU. However, the overlay incurs a high resource footprint of  $\sim 6\text{K}$  DSPs and  $\sim 4552\text{K}$  look-up table (LUT)s on a Virtex UltraScale+ VCU118. Similarly, PP-Trans [8] proposes a custom architecture with pattern pruning support. Pattern pruning removes weights in structured patterns to reduce model size and computation while maintaining hardware-friendly execution. This approach offers up to  $839.7\times$  and  $5.7\times$  speedup over CPU and NVIDIA 3090 GPU, respectively. Despite their performance, both assume fixed encoder-decoder pipelines, resource-rich platforms, and lack flexibility for general-purpose LLM deployment.

To address flexibility, some designs took different approaches, one of which is utilizing pre-defined patterns of the LLM architecture to schedule computations. Khan et al. [30] propose NPE, an end-to-end FPGA accelerator for language models. While it achieves  $35\times$  speedup over CPU, it consumes a large resource footprint of  $\sim 192\text{K}$  LUTs,  $\sim 2\text{K}$  DSPs, and  $\sim 370$  block RAM (BRAM)s on a Zynq Z-7100. Overlay OPU [2], proposed in 2023, improves on NPE with a more efficient processing element (PE) array, modified DSP packing, and FIFO-based data rearrangement, reducing inference latency by  $2.05\times$ . However, it suffers from excessive resource usage  $\sim 4.8\text{K}$  DSPs,  $\sim 1.3\text{K}$  BRAMs, and  $\sim 88\text{K}$  LUTs. It can be scaled down to fit into smaller devices, however, this will negatively impact its tight latency gain. Another promising design, FlexRun [25], dynamically adapts to different model topologies such as BERT and GPT-2 using instruction remapping through compile-time reconfiguration. Compared to baseline FPGAs architectures, FlexRun achieves up to  $2.79\times$  speedup over an NVIDIA V100 GPU, with better architectural flexibility.

Other efforts like HeatViT [15] and Liu et al. [41] extend these ideas with token pruning and quantization, while EdgeLLM [24] demonstrates a heterogeneous CPU-FPGA pipeline. Though effective on high-end hardware, these approaches assume ample dynamic random access memories (DRAMs) and memory bandwidth, making them unsuitable for deployment in edge scenarios.

### 2.2.2 IBERT Accelerator Reference Baseline

As mentioned earlier in this chapter, IBERT is the selected model for evaluating our system. This section reviews a fully parallel IBERT custom accelerator as a reference point for baseline comparison [4]. Fig. 2.1 demonstrates the top-level architecture of the baseline accelerator. The baseline performance was compared to CPU-based inference,

where it was executed on the KV260’s ARM Cortex-A53 processor using a single thread without vector extensions or acceleration libraries. The implementation was run purely on the PS, with no FPGA acceleration or multithreading. The accelerator achieves up to a  $10\times$  speedup over the CPU-based inference. However, it is outperformed by a GPU-based solution using an Nvidia Jetson Orin Nano by a factor of  $2\times$ .

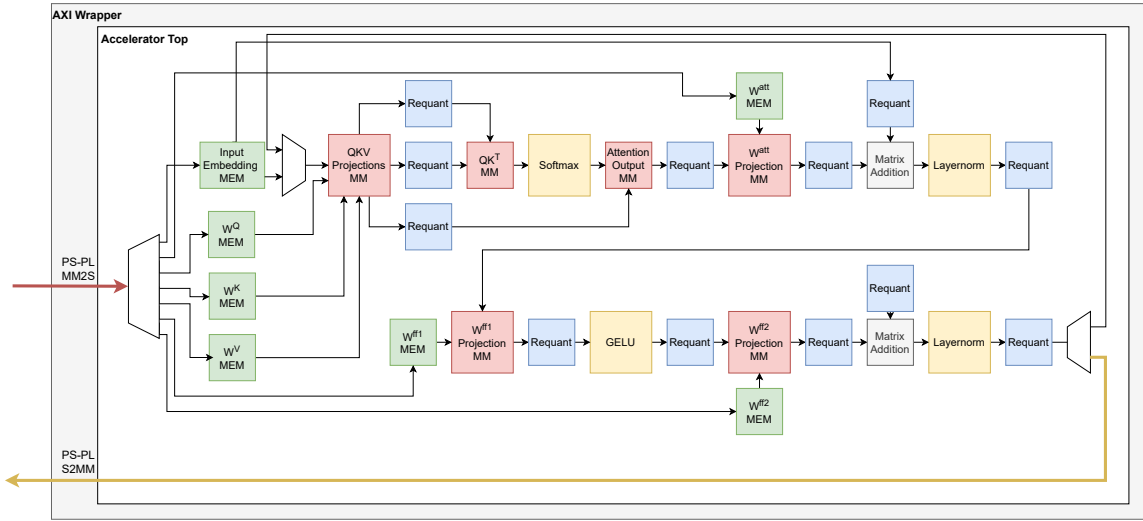


Figure 2.1: High-level architecture of the baseline accelerator. The AXI Wrapper handles the PS–PL AXI4 MM2S/S2MM interfaces and streams data matrices from off-chip memory. The Accelerator Top integrates: A) MM units, B) Matrix addition units, C) non-linear units (Softmax, GELU, LayerNorm), and D) requantization units. Output data is written back through the PS–PL S2MM interface. Colored lines indicate dataflow.

While this custom IBERT accelerator demonstrates the feasibility of integer-only [BERT](#) inference on resource-constrained [FPGAs](#), its design presents critical limitations. The architecture is fully parallel with minimal resource sharing across layers or submodules,

improving performance for a specific model configuration at the expense of flexibility. Nonetheless, due to inefficient resource utilization, lack of memory-bandwidth-aware optimizations, and the absence of fine-grained resource-sharing mechanisms, the accelerator fails to match its GPU counterpart’s performance. This raises broader concerns about the practicality of [FPGA](#)-based acceleration, particularly when GPU offloading offers faster deployment cycles compared to the time-consuming development of cycle-accurate, [FPGAs](#)-compatible designs.

The implementation executes a single statically mapped sequence of operations. The datapath is rigidly defined: memory banks are statically partitioned, and functional units are tightly coupled within a dataflow graph. As a result, adding new functionality, such as decoder support, variant encoder operations, or deeper model layers, would require substantial changes to the datapath and control logic. Moreover, since the design already nears full resource utilization on platforms such as the KV260, expanding the complexity of the model, adding operators, or even increasing the inference sequence length will exceed the available [LUTs](#), [BRAMs](#), or [Xilinx ultra RAM \(URAM\)s](#), requiring the migration to bigger devices. The design also cannot be ported to smaller devices like the PYNQ-Z1 without either slicing the model, which introduces costly off-chip memory transactions, or using very small matrix multipliers, which undermine latency gains and maximize software overheads, where software overhead is the time taken for buffer allocation and [DMA](#) setup. Although the design delivers 18.9 seq./s throughput and 53.5 ms/seq latency for the baseline IBERT configuration, which is 10x speedup over the CPU-based inference, it is poorly suited for future extensions.

A lot of research efforts have been put into [LM](#) inference acceleration on [FPGAs](#) over the last few years. Table 2 and Table 3 summarize a comparison of resource utilization and performance across representative architectures, respectively.

| Year | Framework                 | Device             | Task          | LUT     | DSP  | BRAM/<br>URAM | Lat.  | Thr.<br>( $\times 10^{-3}$ ) |
|------|---------------------------|--------------------|---------------|---------|------|---------------|-------|------------------------------|
| 2020 | FTRANS [33]               | VCU118 (16nm)      | RoBERTa-base  | 452k    | 6531 | –             | 10.6  | 94.2                         |
| 2020 | Multi-Head [43]           | XCVUP13 (16nm)     | MHA & FFN     | 471k    | 129  | 498/–         | 106.7 | –                            |
| 2021 | NPE [30]                  | Zynq Z-7100 (28nm) | BERT-base     | 156k    | 2019 | 562.5/–       | 7.40  | 135.1                        |
| 2021 | Pruning [47]              | Alveo U200 (16nm)  | Custom Trans. | 2364.5k | 6840 | –             | 10.35 | 966.2                        |
| 2022 | DFX [23]                  | Alveo U280 (16nm)  | GPT2-1.5B     | 520k    | 3533 | 1192/104      | 47.6  | 21 <sup>*</sup>              |
| 2023 | FET-OPU [2]               | Alveo U200 (16nm)  | RoBERTa-base  | 886k    | 4864 | 1357/–        | 6.82  | 146.6                        |
| 2023 | PP-Trans [8]              | Alveo U200 (16nm)  | BERT-base     | 83k     | 2560 | 58/–          | 1.27  | 787.4                        |
| 2024 | ODE [46]                  | ZCU104 (16nm)      | CNN-Trans.    | 66k     | 206  | 595/46        | 162.1 | 6.2                          |
| 2024 | FLightLLM [59]            | Alveo U280 (16nm)  | LLaMA2-7B     | 574k    | 6345 | 1252/792      | 18.2  | 55 <sup>*</sup>              |
| 2024 | LlamaF [56]               | ZCU102 (16nm)      | TinyLLaMA     | 164k    | 528  | 223/–         | 666.7 | 1.5 <sup>*</sup>             |
| 2025 | Li et al [34]             | KV260 (16nm)       | LLaMA2-7B     | 78k     | 291  | 36.5/–        | 204.0 | 4.9 <sup>*</sup>             |
| 2025 | EdgeLLM [24]              | Alveo U280 (16nm)  | ChatGLM-6B    | 967k    | 5587 | 1734/–        | 13.3  | 75 <sup>*</sup>              |
| 2025 | IBERT <sub>baseline</sub> | KV260 (16nm)       | RoBERTa-base  | 69k     | 1123 | 105/43        | 53.5  | 18.9                         |

Table 2: Breakdown of state-of-the-art FPGA-based language models inference accelerators. Lower latency (Lat.) is better. Higher Throughput (Thr.) is better.\* Decoder-based models’ throughput is tokens/s.

### 2.2.3 General-Purpose Accelerators on CPUs and GPUs

GPU-based accelerators focus primarily on kernel fusion and memory traffic reduction. Choi et al. [11] recompose the [softmax function \(Softmax\)](#) layer by fusing its pre- and post-processing stages, yielding  $1.12\text{--}1.65\times$  speedups across [BERT](#) [14], GPT-Neo [7], Big-Bird, and Longformer. LLMA [57] introduces reference-guided token copying to reduce redundant generation, achieving  $2\times$  speedup over CPU baselines when significant output overlap is detected. This fusion technique is orthogonal to our work but underscores the importance of minimizing auxiliary overhead—an objective similarly tackled by our hardware resource manager for [NL](#) operations. UltraFastBERT [5], a CPU-based accelerator, replaces standard feed-forward layers with fast sparse approximations, selectively activating a subset of neurons per layer and attaining up to  $78\times$  speedup over standard [FFN](#) and  $40\times$  speedup in PyTorch implementations. Although effective, this approach requires

dynamic runtime support, which is impractical on [FPGAs](#).

#### 2.2.4 ASIC-Based Accelerators

ASIC accelerators achieve high efficiency through approximation and structured sparsity. ELSA [21] and A3 [20] propose approximate similarity estimation in attention mechanisms, yielding  $157\times$  speedup and over  $1000\times$  energy efficiency gains versus V100 and Volta GPUs, respectively. SpAtten [54] applies cascade token and head pruning, achieving  $347\times$  speedup over CPU and  $4059\times$  energy savings over Titan GPU. Sanger [42] employs runtime sparsity prediction and a reconfigurable systolic array, reporting  $22.7\times$  acceleration over dense CPU inference. Energon [61] introduces multi-round mixed-precision filtering, reaching  $168\times$  speedup and  $10,000\times$  energy efficiency over CPU. Although the high latency gains, these designs lack reconfigurability and require extensive chip-level design. In contrast, our work leverages [FPGAs](#) reconfigurability to provide flexible support for integer-only [LLM](#) inference without requiring custom silicon or fixed function blocks.

Moreover, ASIC accelerators often target decoder-based [LLMs](#) or massive batch sizes to amortize hardware cost, which limits their usability in low-latency, single-batch inference.

#### 2.2.5 In-Memory Accelerators

In-memory accelerators use near-data compute principles to reduce transfer overhead. ATT [18] uses [resistive random access memories \(ReRAMs\)](#) crossbars for matrix-vector and [Softmax](#) operations, achieving  $202\times$  speedup over GPU. ReTransformer [58] integrates decomposition and in-memory [Softmax](#) logic for a  $23.2\times$  speedup and  $1086\times$  energy savings compared to NVIDIA TITAN RTX GPU. iMCAT [32] targets long-sequence tasks with [content](#)

addressable memory (CAM) based filtering and caching, yielding  $200\times$  speedup and  $41\times$  energy efficiency. X-Former [49] separates static and dynamic operations across non-volatile memory (NVM) and complementary metal–oxide–semiconductor (CMOS) tiles, achieving  $85\times$  latency reduction and  $7.5\times$  energy savings. Flash [1] schedules inference with flash-DRAMs paging and reuse-aware memory access, delivering up to  $25\times$  speedup over naive CPU memory loading. These approaches achieve high energy efficiency but are not viable for edge-class FPGAs due to the lack of NVM support and inflexibility to accommodate changing model architectures. Our work, while not in-memory, achieves efficient data locality using resource sharing and AXI-compatible streaming modes—mimicking some benefits of near-data compute while remaining general-purpose.

ASIC, in-memory, and flash-based accelerators tend to provide much higher speedup compared to FPGAs or GPU-based accelerators [27]. However, they are excluded from this thesis due to their design inflexibility, higher development complexity, and lack of deployment on low-cost platforms. Table 3 summarizes a comparison of performance across representative architectures.

| Year | Framework                 | Tech          | Speedup     | Eff.         | Baseline         |
|------|---------------------------|---------------|-------------|--------------|------------------|
| 2019 | MNNFast [26]              | FPGA 7020     | 5.4×        | 6.5×         | CPU (Xeon)       |
| 2020 | FTRANS [33]               | FPGA VCU118   | 27–81×      | 8.8×         | CPU/GPU RTX5000  |
| 2020 | Multi-Head [43]           | FPGA XCVUP13  | 14×         | –            | GPU V100         |
| 2021 | NPE [30]                  | FPGA Zynq7100 | 35×         | 4–6×         | CPU/GPU RTX5000  |
| 2021 | Pruning [47]              | FPGA U200     | 11×/2×      | –            | CPU/GPU TX2      |
| 2022 | DFX [23]                  | FPGA U280     | 3.8×        | 4×           | GPU V100         |
| 2023 | FET-OPU [2]               | FPGA U280     | 15×/2.9×    | –            | CPU/GPU 3090     |
| 2023 | FlexRun [25]              | FPGA S10      | 2.7×        | –            | GPU V100         |
| 2023 | PP-Transformer [8]        | FPGA U200     | 5.7×/146.7× | –            | CPU/GPU 3090     |
| 2024 | ODE [46]                  | FPGA ZCU104   | 12.8×       | 9.2×         | ARM A53          |
| 2024 | FlightLLM [59]            | FPGA U280     | 1.2×        | 6.0×         | GPU A100         |
| 2025 | Li et al [34]             | FPGA KV260    | 0.3×        | –            | Jetson Orin Nano |
| 2025 | IBERT <sub>baseline</sub> | FPGA PYNQ-Z1  | 0.03×       | –            | Jetson Orin Nano |
| 2025 | IBERT <sub>baseline</sub> | FPGA KV260    | 0.5×        | 0.9×         | Jetson Orin Nano |
| 2022 | SoftMax [11]              | GPU A100      | 2.5×        | –            | GPU A100         |
| 2023 | LLMA [57]                 | GPU V100      | 2×          | –            | GPU V100         |
| 2023 | UltraFastBERT [5]         | CPU           | 78×         | –            | CPU              |
| 2020 | A3 [20]                   | ASIC 40nm     | 7×          | 11×          | CPU/GPU Volta    |
| 2021 | ELSA [21]                 | ASIC 40nm     | 157×        | 1265×        | GPU V100/TPU     |
| 2021 | SpAtten [54]              | ASIC 40nm     | 347×/162×   | 4059×/1093×  | CPU/GPU Titan    |
| 2021 | Sanger [42]               | ASIC 55nm     | 22.7×/4.64× | –            | CPU/GPU V100     |
| 2023 | Energon [61]              | ASIC 45nm     | 168×/8.7×   | 10000×/1000× | CPU/GPU V100     |
| 2020 | ATT [18]                  | In-mem        | 202×        | –            | CPU/GPU Volta    |
| 2020 | ReTransformer [58]        | In-mem        | 23×         | 1086×        | GPU              |
| 2022 | iMCAT [32]                | In-mem        | 200×        | 41×          | GPU Titan RTX    |
| 2023 | X-Former [49]             | In-mem        | 202×        | 7.5×         | GPU GTX1060      |
| 2023 | Flash [1]                 | Flash         | 25×         | –            | CPU/GPU          |

Table 3: Extended breakdown of language models accelerators. The original table is taken from [27]. Higher energy efficiency and speedup are better.

## 2.3 Algorithmic Methods for Efficient Inference

Quantization and pruning have been critical approaches to reduce both the memory footprint and computational requirements of large models. Some pruning techniques explored in the literature include: structured [45], semi-structured [50], and unstructured pruning [38]. Similarly for quantization, there have been efforts in [post-training quantization](#)

(PTQ), [55, 31], and quantization-aware training (QAT) [12]. Recent efforts have also been exploring the use of approximate arithmetic for addressing the increased computational demands of the transformer-based models [13]. For simplicity, we focus only on QAT techniques, and disregard weights and/or activation pruning algorithms. Early research on quantization—primarily in the context of CNN [40, 60]—demonstrated that representing weights and activations with low precision can lead to substantial reductions in model size and computational requirements. For instance, 8-bit integer (INT8) quantization typically reduces model size by approximately  $4\times$  compared to FP32 models. Additionally, it can decrease computational cost and power consumption by around 40% [40, 60]. However, applying these methods to transformer models introduces additional challenges due to NL functions such as rectified linear unit (ReLU), gaussian error linear unit (GELU), Softmax, and layer normalization (LayerNorm). Many early transformer quantization approaches adopted simulated quantization, in which parameters are stored in low precision while key NL operations are performed using floating point arithmetic [29, 6]. This hybrid approach does not fully reap the benefits of integer arithmetic and adds overhead through dequantization and re-quantization steps. A different approach is presented in [31]. IBERT proposes a complete integer-only quantization scheme for transformer models such as BERT by developing integer-only approximations for NL functions. For instance, IBERT introduces i-GELU, which uses a second-order polynomial approximation to closely mimic the original GELU function with an average error as low as  $8.2 \times 10^{-3}$  compared to the original FP32 GELU, demonstrating a close approximation with minimal accuracy degradation. Similar integer only methods are applied to Softmax and LayerNorm, enabling the entire inference process to be executed using INT8 for multiplications and 32-bit integer (INT32) for accumulations. This approach not only reduces memory bandwidth and power consumption but also makes the design compatible with processors that have limited or

no floating point support.

## 2.4 On-Chip Memory Management

Memory management is one of the most critical challenges in recent [LLM](#) accelerators, as these models feature billions of parameters and large data transfers that demand high memory capacity and bandwidth. Efficient utilization of on chip memory and high memory bandwidth are paramount for FPGA-based acceleration—especially on embedded devices with very limited memory capacity. However, on-chip memory management is often discussed as part of the overall dataflow and scheduling strategy rather than as a standalone focus. A key memory management strategy that was discussed in modern accelerators is the optimization of data partitioning and burst transfer schemes. For example, systems like FlightLLM [\[59\]](#) and DFX [\[23\]](#) organize large weight matrices and intermediate data into contiguous blocks that match the bus width of the [HBM](#), thereby enabling large, consecutive burst transfers. Such arrangements maximize the effective bandwidth and reduce the number of costly off-chip accesses. However, these techniques often require complex data arrangement formats and multiplexing logic, which are manageable on cloud-class [FPGAs](#) equipped with [HBM](#) but pose challenges on embedded systems with limited resources [\[59, 23\]](#). Due to the abundance of on-chip memory resources, these approaches also don't put high emphasis on the on-chip memory utilization or resources. They rely on a continuous off-chip data stream through a direct interface between the [processing system \(PS\)](#) and [programmable logic \(PL\)](#).

Several other [FPGA](#)-based accelerators target transformer inference through [MM](#) multiplication optimization and memory bandwidth reduction. For instance MNNFast [\[26\]](#) introduces a column-based streaming architecture with zero-skipping and embedding cache

optimizations, achieving up to  $6.5\times$  energy efficiency over a CPU baseline, but a limited  $5.4\times$  speedup over CPU. However, its reliance on embedding caches underscores the importance of efficient on-chip memory usage. Lu et al. [43] propose a systolic-based accelerator for the two most computationally intensive parts of the transformer network, multi-head attention and FFN blocks, sharing resources between the two while optimizing NL functions such as Softmax and LayerNorm. Although the system demonstrates a  $14.6\times$  speedup of multi-head attention computation over an NVIDIA V100 GPU, it uses a cloud-class board with  $\sim 471\text{K}$  LUTs,  $\sim 500$  BRAMs, and  $\sim 129$  DSPs utilization. Their approaches are tailored for cloud-class FPGAs with abundant BRAM. In contrast, our design targets memory-constrained devices and uses a flexible memory manager to minimize off-chip access.

Although prior works such as the one proposed by Li, J. et al [34] have employed specialized memory-management dataflow to approach near-peak memory bandwidth with an impressive token throughput of 4.9 tokens/s on a KV260 board, they rely on complex data multiplexing on bare-metal to enable large burst transfers from off-chip memory. This design is customized to the architecture they built, which makes it unscalable.

Our design takes a different approach by leveraging the regularity inherent in transformers’ operations. The runtime for multi-head attention is overwhelmingly dominated by matrix computations with these computations accounting for more than 97% of the total runtime [56]. Accordingly, we target maximizing the size of the systolic array MM multiplication architecture, and we statically schedule the routings of the on-chip random access memories (RAMs) connected to the interface of this systolic array. Pushing the computation capabilities of the target platforms to the limit. In our system, weights and activations are partitioned into shreds and blocks directly fed to the FPGA through separate AXI ports—one for each shred. Leveraging the symmetry of the systolic multiplier

architecture. The statically managed architecture offers several critical advantages over the literature, including:

- **Flexibility:** The modular design of the double-buffered systolic [MM](#) multiplier allows the [MNGR](#) architecture to scale easily to different systolic array sizes and adapt to varying resource constraints without requiring redesign. Unlike the typical, more rigid accelerator architectures, each of the systolic frontier I/O is connected to one or more [BRAMs/URAMs](#) that are configured by the control [finite state machine \(FSM\)](#).
- **Deterministic Scheduling:** The rhythmic, pipelined data flow of the systolic interface memory layout enables the design of a fine-grained, deterministic on-chip memory scheduling mechanism. Ensuring on chip [BRAMs/URAMs](#) are efficiently re-used and that off-chip memory access is minimized.
- **Simplified Data Management:** By dividing weights into shreds and blocks and feeding each shred through a dedicated [AXI](#) port, our design obviates the need for complex multiplexing schemes, thereby reducing hardware overhead and latency.

Even though cloud-class systems achieve impressive throughput through extensive parallelism, they are not applicable to real-world scenarios. Expensive server-class [FPGAs](#) are expected to handle multiple concurrent user inquiries, whereas single-batch inference is more commonly seen in edge environments. To the best of our knowledge, there has been no focus in the existing designs on addressing the deployment of scheduled **cycle-accurate** language models accelerators on edge-class [FPGAs](#) with an emphasis on on-chip memory utilization and minimal off-chip bandwidth demand. In the next chapter, we break down the model chosen for evaluation, the target [FPGA](#) devices, and outline the model deployment strategy.

# Chapter 3

## Target Model and Deployment Strategy

This chapter introduces the evaluation model and outlines the deployment optimizations for accelerating its inference on resource-constrained [FPGAs](#). It begins with a structural overview of the model acceleration flow and the rationale behind device selection. It then describes key platform constraints and presents the core design strategy and top-level system architecture that guide the implementation detailed in [Chapter 4](#).

### 3.1 Target Model and Hardware

Among the various transformer-based models available, IBERT has been chosen as the reference model for this work. Unlike traditional [BERT](#) [\[14\]](#) implementations that rely on floating point computations, IBERT is optimized to operate entirely with integer arithmetic, making it an ideal candidate for [FPGA](#)-based deployment. IBERT structured architecture is also nearly identical to other encoder-only transformers, like I-ViT, I-DeiT [\[37\]](#), making it an excellent test case for evaluating memory management strategies and

computation scheduling on embedded [FPGAs](#) hardware. To accurately showcase IBERT deployment, two [FPGA](#) platforms have been considered: the PYNQ-Z1 and the KV260. The following sections provide a detailed breakdown of the IBERT model and the hardware-specific challenges associated with [FPGA](#)-based acceleration. The focus will be on developing an efficient resource manager that minimizes off-chip memory access, optimizes data reuse, and ensures seamless integration of transformer-based models, emphasizing [BERT](#) [14], within the edge-class [FPGA](#) fabric.

### 3.1.1 IBERT Model Overview

IBERT is a specialized adaptation of the [BERT](#) [14] model, designed to operate entirely using integer arithmetic. Standard transformer-based models like [BERT](#) rely heavily on floating-point operations, which pose significant challenges for energy-efficient and resource-constrained hardware platforms. IBERT addresses this using integer-based approximations for [NL](#) functions such as [GELU](#), [Softmax](#), and [LayerNorm](#). The model executes all matrix multiplications using [INT8](#) operands with [INT32](#) accumulation before re-quantization to [INT8](#) again, ensuring computational efficiency without sacrificing numerical stability. IBERT shows comparable or slightly higher accuracy than the [FP32](#) baseline. Table 4 demonstrates the quantization results on [General Language Understanding Evaluation \(GLUE\)](#) benchmark.

| Model       | Precision | MNLI-m<br>(%) | QQP<br>(%) | QNLI<br>(%) | SST-2<br>(%) | CoLA<br>(%) | STS-B<br>(%) | MRPC<br>(%) | RTE<br>(%) | Avg.<br>(%) |
|-------------|-----------|---------------|------------|-------------|--------------|-------------|--------------|-------------|------------|-------------|
| I-BERT [31] | FP32      | 87.8          | 90.4       | 92.8        | 94.6         | 61.2        | 91.1         | 90.9        | 78.0       | 86.0        |
| I-BERT [31] | INT8      | 87.5          | 90.2       | 92.8        | 95.2         | 62.5        | 90.8         | 91.1        | 79.4       | 86.3        |

Table 4: GLUE benchmark results of IBERT using FP32 and INT8 precisions. Results were obtained from [31]. Higher scores are better.

The IBERT-base model preserves the core architecture of BERT-base but integrates several modifications that enhance its efficiency for integer-based computation, which makes it a perfect choice to showcase our design. The model consists of an embedding layer, followed by 12 transformer encoder layers, and a LM head layer. Totaling around 125 million parameters.

### 3.1.2 Model Deployment

To deploy the model on our accelerator, we process the first and last layers on the CPU. Due to the large memory footprint required to store the embedding matrix and its simple lookup behavior, it is resource-wasteful to accelerate them on the PL. These layers account for over 30% of the model weights and exhibit poor compute-to-memory ratios. Their memory-dominated behavior would activate over 250 BRAMs and introduce significant routing complexity with minimal DSP utilization, making them inefficient to accelerate on an FPGA. Moreover, the required memory footprint exceeds the capacity of most FPGA platforms. Offloading these layers to the CPU avoids this overhead by leveraging the abundantly available off-chip memory and better-suited memory access patterns. The flow of IBERT inference on our architecture is shown in Fig. 3.1. The 12 encoder layers are accelerated on the device. Each encoder layer includes 12 self-attention sublayers followed by FFN and LayerNorm operations. The multi-head attention mechanism allows the model to consider multiple contextual perspectives of a word in a sentence. LayerNorm normalizes activations using integer-only approximations. The FFN applies a NL transformation to the contextual embeddings, further projecting and refining them.

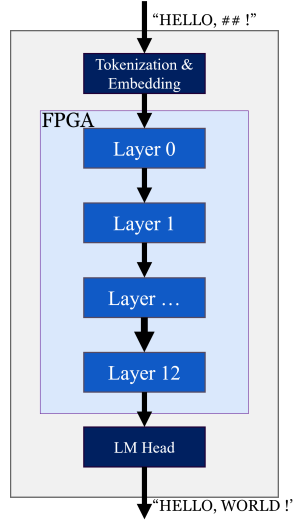


Figure 3.1: Simplified flow of IBERT-base [31] inference on our architecture. The input is tokenized and embedded on the CPU, processed by the encoder layers on the FPGA, and decoded by the LM head on the CPU to generate the output.

The encoder module sub-layers can be modularized as: [IBertAttention sub-layer \(ATTN\)](#), [IBertSelfOutput sub-layer \(SELFOUT\)](#), [IBERTIntermediate sub-layer \(INTER\)](#), and [IBERTOutput sub-layer \(OUTPUT\)](#). Demonstrated in Figures 3.2, 3.3, 3.4, and 3.5 respectively. A full IBERT encoder layer is the concatenation of these blocks in order. Other quantized encoder-only models, such as IViT and IDieT [37], follow the exact same flow with minor differences in the encoder layer operation order.

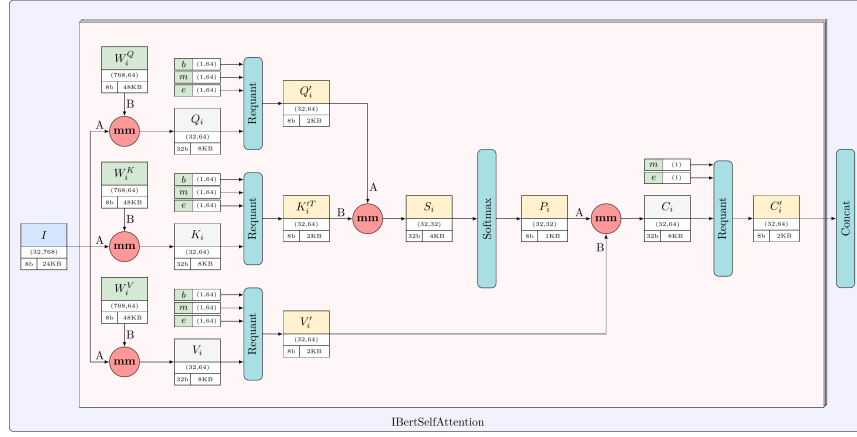


Figure 3.2: Self-Attention block diagram. Input embeddings are projected to quantized queries, keys, and values. Attention scores are computed and passed through softmax to generate context vectors.

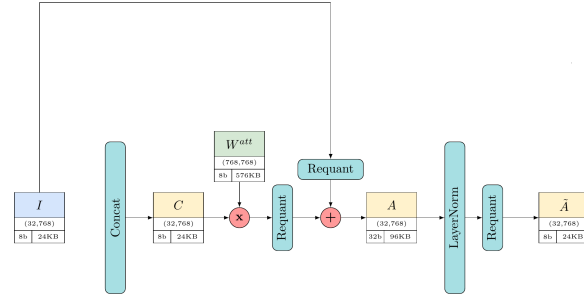


Figure 3.3: Self-out sublayer block diagram. Context vectors are multiplied with the projection matrix, scaled, and clamped to `INT32`. The residual input is scaled and clamped to `INT32`. The two matrices are then added, normalized, and quantized `INT8`.

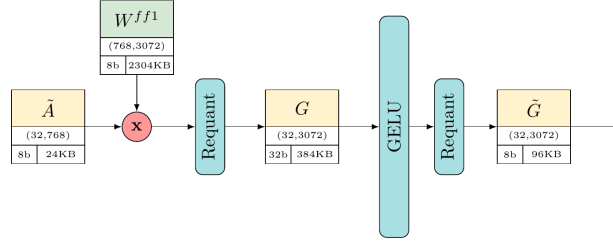


Figure 3.4: Intermediate sublayer block diagram. Linear projection of normalized outputs followed by requantization and GELU activation.

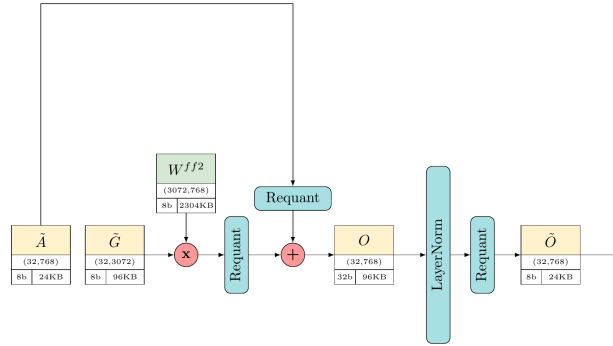


Figure 3.5: Layer-out sublayer block diagram. Activated representations are projected, scaled, and clamped to `INT32`. Then, it is combined via scaled `INT32` residual addition, normalized, and quantized to `INT8`.

\*Note: IBERT Fig. 3.2, 3.3, 3.4, and 3.5 are adopted from the ECE 327/627 course lab manuals.

### 3.1.3 FPGA Platform Selection

Selecting an appropriate `FPGA` platform for showcasing our design requires consideration of memory availability, `DSP` count, and on-chip interconnect bandwidth. Two primary `FPGAs` are evaluated for deployment: The PYNQ-Z1 and Kria KV260.

The PYNQ-Z1 board is a low-cost, widely available [FPGA](#) platform that serves as an ideal choice for showcasing the capabilities of our design. It features an ARM Cortex-A9 processor, enabling hybrid CPU-[FPGA](#) execution for tasks such as model partitioning and data preprocessing. However, its limited [BRAM](#) capacity of 630 KB severely restricts its ability to accelerate language models in a competitive latency. Additionally, the lack of a high-speed [DDR](#) interface makes off-chip memory access costly, further constraining performance. As a result, while the PYNQ-Z1 can be useful for showcasing the packing abilities of the memory manager, it is not suitable for full-scale deployment compared to its slightly more expensive competitors. The KV260, in contrast, offers significantly greater hardware resources, making it a more viable platform. It provides a larger number of [BRAMs](#) and [DSP](#) slices, enabling deeper pipeline structures and larger systolic multipliers. The KV260 also supports high-speed [AXI4](#) datastreams, allowing for efficient [DDR](#) memory access and minimizing latency associated with off-chip communication. Table 5 demonstrates a summary of PYNQ-Z1 and KV260 boards specifications.

| Specifications/Device            | PYNQ-Z1                             | KV260                               |
|----------------------------------|-------------------------------------|-------------------------------------|
| <b>Architecture</b>              | Zynq-7000 (28nm)                    | Zynq UltraScale+ (16nm)             |
| <b>BRAM</b>                      | 630 KB: 140 x 36Kb or<br>280 x 18Kb | 648 KB: 144 x 36Kb or<br>288 x 18Kb |
| <b>URAM</b>                      | –                                   | 2.25 MB: 64 x 288Kb                 |
| <b>DDR</b>                       | 500 MB                              | 4 GB                                |
| <b>LUT</b>                       | 71K                                 | 175K                                |
| <b>FF</b>                        | 106K                                | 234K                                |
| <b>DSP Slices</b>                | 220 DSP48E1                         | 1081 DSP48E2                        |
| <b>PS-PL AXI Bandwidth</b>       | 32b                                 | 32–128b                             |
| <b>PS-PL AXI Interface Count</b> | 1                                   | 1–4                                 |
| <b>CPU Frequency</b>             | 650 MHz                             | 1.33 GHz                            |

Table 5: Comparison of resources between PYNQ-Z1 and KV260 boards.

## 3.2 Platforms Constraints, and Acceleration Strategy

Deploying an encoder-based language model on the selected [FPGAs](#) introduces a range of architectural constraints that must be addressed to ensure optimal performance. The selected [FPGAs](#) provide limited on-chip memory, primarily in the form of [BRAMs](#) or [URAMs](#), which necessitate careful memory management and scheduling. In general, the on-chip memory capacity of edge-class [FPGAs](#) are not sufficient for a fully parallel [LLM](#) deployment and acceleration.

To optimize execution, a hierarchical memory structure is implemented. First-level storage, consisting of [BRAMs](#), is used for high-speed access to frequently needed tensors. Second-level storage, which relies on [DDR](#) DRAM, contains all the weights and activations that are streamed to the on-chip memories. Additionally, dedicated memory blocks are allocated for [NL](#) units vector storage, ensuring that matrix-vector computations are not blocked by other matrix-matrix operations executing in parallel. The top-level system architecture of both devices is shown in [3.6](#).

Beyond memory constraints, encoder-based models' execution relies on data movement between the [FPGA](#)'s compute units and memory subsystem. Excessive off-chip memory access can significantly degrade performance due to high access latency and software overhead introduced by [DMA](#) setup and buffer allocation. To mitigate this, the resource manager is designed to minimize off-chip communication, restricting reads and writes to, at most, one per inference. Techniques such as data prefetching and double buffering are employed to hide memory latency, allowing computation and memory transfers to overlap seamlessly.

Address generation logic further optimizes read-and-write access by supporting structured access patterns such as row-wise, cyclic, and systolic, which improve data locality

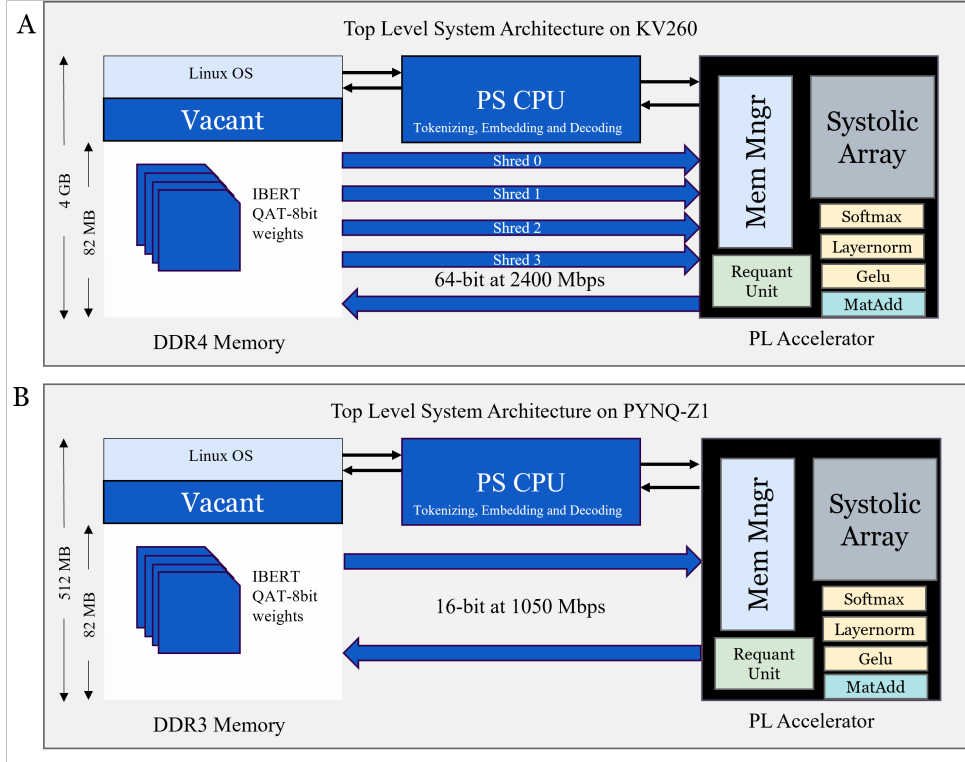


Figure 3.6: Top Level System Architecture on: A) KV260 with a total bandwidth of 64-bit at 2400 Mbps or 19.2 GB/s divided across 4 DMA channels. B) PYNQ-Z1 with a total bandwidth of 16-bit at 1050 Mbps or 2.1 GB/s.

and reuse. The read-and-write access patterns will be discussed in Chapter 4.

Another key challenge is balancing computational parallelism with resource utilization. FPGAs enable customizable parallelism but are constrained by available LUTs, DSP slices, and BRAMs/URAMs. Achieving high throughput necessitates careful design choices. Instead of fully unrolling transformer layers, which would consume excessive FPGA resources, an optimized FSM statically schedules the coarse operations, reducing redundant resource allocation. The target is to utilize the maximum computation resources that give a perfor-

mance balance with the memory bandwidth limit ( $\leq 19.2$  GB/s). In the next chapter, we detail the hardware architecture and optimization techniques of the full accelerator with our proposed resource management infrastructure.

# Chapter 4

## Hardware Architecture

The hardware architecture of the [LM](#) accelerator with resource scheduling support we proposed in [Chapter 3](#) consists of five main components: The Systolic [MM](#) multiplier, the [MNGR](#), the [requantization unit \(RU\)](#), the [matrix addition unit \(MATADD\)](#), and the [non-linear processing unit \(NLU\)](#), as shown in [Fig. 4.2](#). In this design, the [MNGR](#) functions as the central streaming unit between the CPU, memory banks, and accelerator blocks. Some of these components have been designed or optimized to operate with the [MNGR](#), and some have been imported from the fully parallel design for simplicity. We will introduce the former components in the following subsections.

### 4.1 Systolic Matrix-Matrix Multiplication

The hardware architecture integrates a systolic [MM](#) multiplication engine that operates in conjunction with the structured memory subsystem. The systolic array is a two-dimensional grid of [PEs](#) that execute [multiply and accumulate \(MAC\)](#) operations in a

pipelined fashion. Matrices A ( $M1 \times M2$ ) and B ( $M2 \times M3$ ) are streamed into the systolic array to produce matrix D ( $M1 \times M3$ ). The systolic array has dimensions  $N \times N$ , meaning that at any given moment,  $N$  rows of A and  $N$  columns of B are being processed. Accordingly, a number of [BRAM](#) blocks are connected to each port of the systolic array based on  $M1/N$  and  $M3/N$  parameters to load and store the data. More details about the memory layout will be discussed later in this chapter. Systolic data partitioning, streaming dataflow, and multiplication algorithm are discussed below, assuming [MM](#) multiplication operation is  $D = A \times B$ .

- **Systolic Data Partitioning:**

- For cases where  $M1 > N$  or  $M3 > N$ , the computation is staggered into multiple phases as shown in [Fig. 4.1](#).
- Each phase handles a sub-matrix of A and B.
- When  $M3 \gg N$ , we use double buffering to fully multiply A with a block of B as shown in [Fig. 4.1](#). A “block” refers to a column-wise partition of matrix B. The only constraints are that the block width must be an integer multiple of  $N$ .
- The A matrix, proportional to the context length, is much smaller than the B matrix (proportional to the weight matrices). Hence, there is no need for a row-wise split or double buffering of matrix A.

- **Streaming Dataflow:**

- Matrix A is fed row-wise into the systolic array. Matrix B is fed column-wise.
- A and B multiplication operands require custom loading and storing logic that supports both double buffering and phasing.

- PE initiation of final sums propagates diagonally, signaling the end of a series of partial sums. This allows continuous input data streaming.
- Final computed results for D exit from the rightmost column. However, they also need custom double-buffered load/store logic.
- Because a block is a full column set of the B matrix, results are computed block-wise as shown in Fig. 4.1. A dedicated control FSM controls the computation flow of the multiplier based on the central MNGR FSM. This optimization removes the need to store the entire B matrix on-chip. Only a maximum of two limited-size blocks need to be stored at once to ensure full-throughput computation.

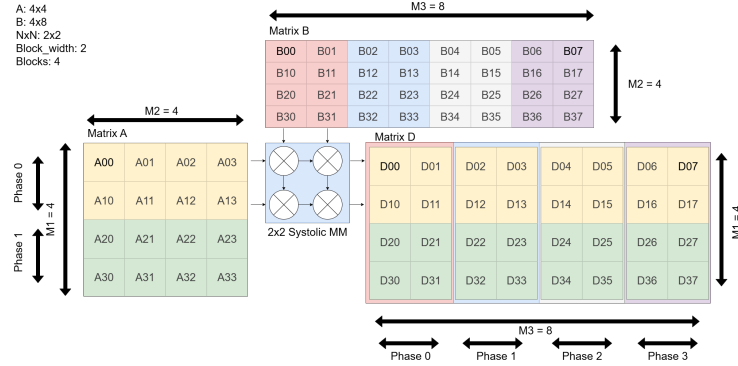


Figure 4.1: Phased execution of a  $2 \times 2$  systolic multiplier with double buffering. Matrix  $B$  is divided into  $M_3/N$  slices of size  $M_2 \times N$ , each reused across all  $M_1/N$  tiles of matrix  $A$  of size  $N \times M_2$ . This enables overlapping phases (e.g., yellow  $\times$  red, green  $\times$  red) and interleaved computation of matrix  $D$  without stalls. Color coding shows operand-to-result mapping per phase.

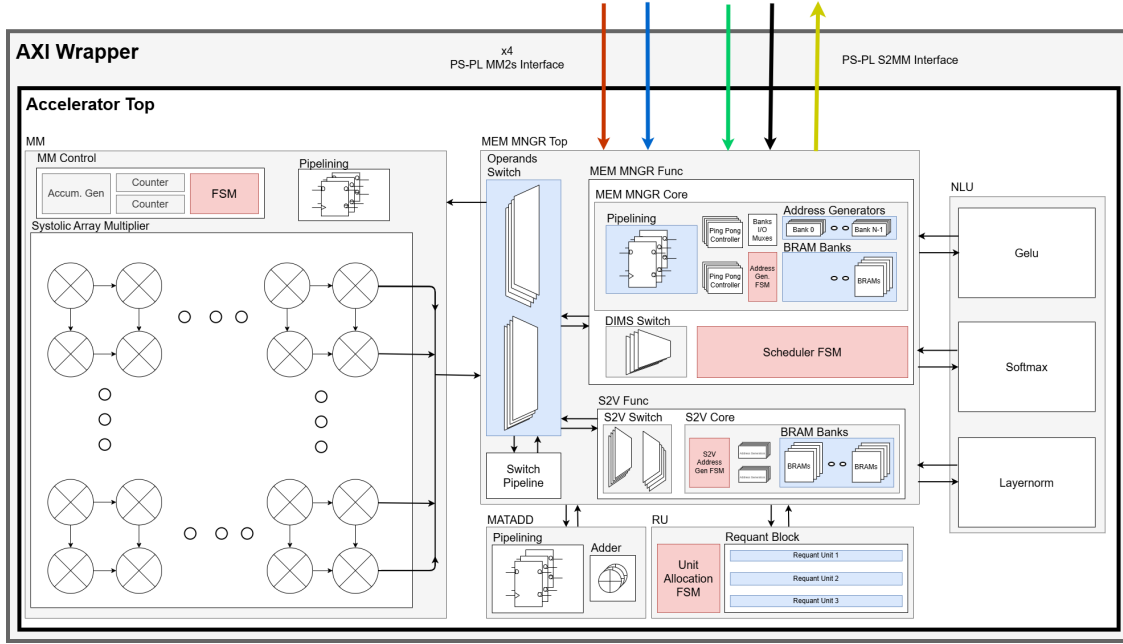


Figure 4.2: High-level architecture of the accelerator on KV260. The AXI Wrapper (top) handles up to four parallel AXI4 MM2S interfaces, managing the accelerator requests, and distributing input data from multiple memory maps to the **MNGR** block. The Accelerator Top (bottom) includes: A) **MNGR** for scheduling and memory coordination, B) Systolic **MM** unit, C) **RU**, and D) **NLU** with **GELU**, **Softmax**, and **LayerNorm**. Output data is written back via the PS-PL S2MM interface. Colored lines indicate dataflow.

## 4.2 Resource Scheduling

The **MNGR** is the central unit for data movement across the accelerator. Unlike conventional FPGA designs that rely on spatially distributed on-chip memory blocks, our architecture centralizes memory access through a unified **MNGR**. This choice simplifies control logic, enables structured scheduling for systolic **MM**, and leverages the **FPGA**'s

abundant interconnect and LUT resources for efficient timing, routing, and pipelining. Centralization also enhances portability, allowing the design to scale across FPGA platforms without re-architecting memory placement. This centralized approach allows us to maximize compute unit instantiation. As the memory banks will be concatenated and wrapped around the compute unit that reports the most activity. In our case, the goal was to scale the systolic array to the full capacity permitted by the device’s DSP and BRAM budgets, while easily maintaining timing closure through resource-cheap pipelining. Thus, increasing throughput and maximizing DDR bandwidth utilization with minimal control overhead.

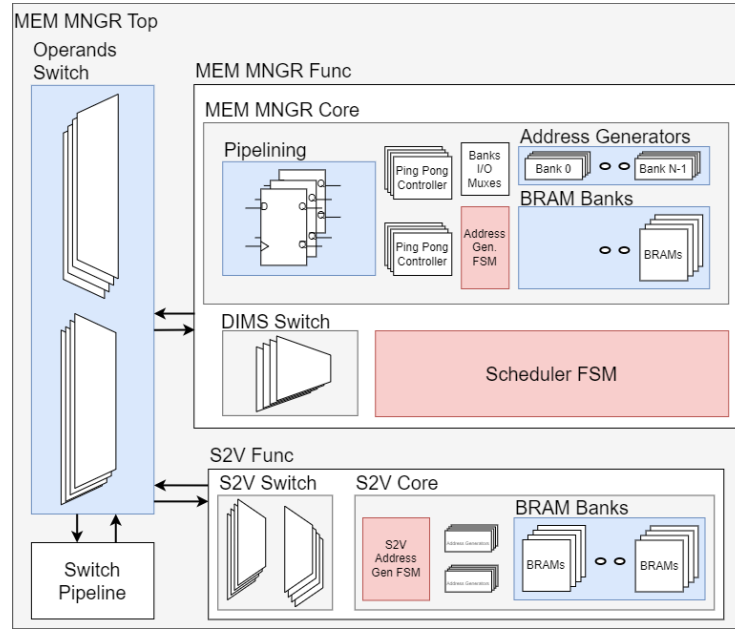


Figure 4.3: Memory manager top module.

The subsystem, demonstrated in Fig. 4.3 is modularized into four principal components: The schedule, the MNGR core and stream-to-vector (S2V) manager, and switches.

### 4.2.1 Phase Scheduler and Control Synchronization

The scheduler implements the central **FSM** that governs the operational sequencing of the **MNGR**. It defines transitions through distinct execution phases: loading operands, initiating systolic **MM**, issuing output writeback, loading post-processing vectors for **NL** units, storing weights and activations, controlling I/O switches, and managing **AXI** transactions. These transitions are determined by handshake signals from the **AXI** interfaces, internal flags from the systolic computation block, and valid-ready events from buffer state monitors.

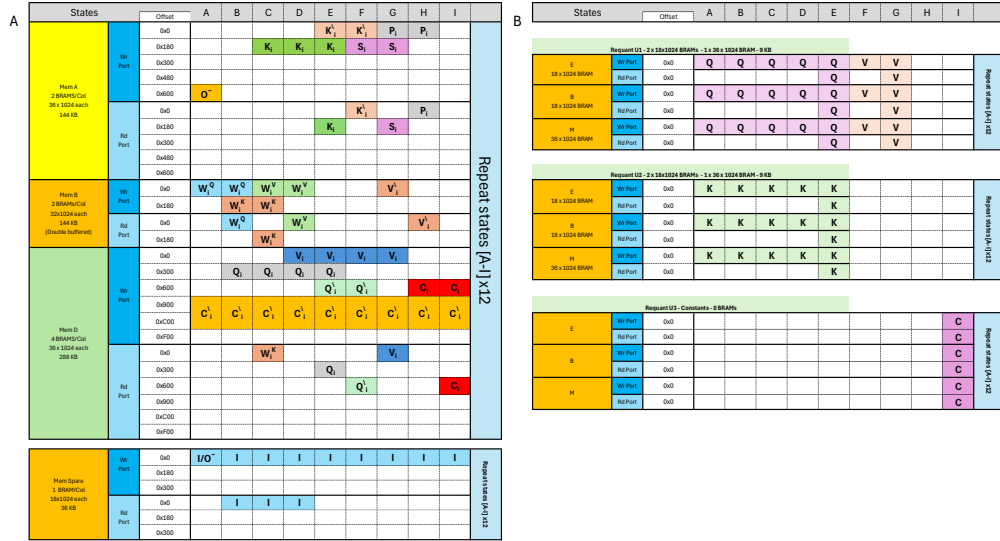


Figure 4.4: Memory access schedule for the IBERT self-attention sub-layer. A) Weights  $W^Q, W^K, W^V$  are stored in Mem B; activations are stored in Mem Spare; and intermediate results are stored in Mem A and Mem D. B) NL vectors are stored in Requant U1–U3 at states A, F, and I; loaded at states E, G, and C. States A–I are repeated for the 12 heads.

By providing a global synchronization set, this design ensures deadlock-free operation, enables dynamic slicing of computation, and prevents race conditions across multiple asynchronous subsystems in the overlay. Its deterministic phase control is crucial for pipelined and overlapped execution modes. In other words, when operand streaming, systolic accumulation, and [NL](#) processing must all coexist on limited on-chip resources simultaneously.

[Fig. 4.4](#) presents a simplified tabular schedule that governs the data flow of the IBERT [ATTN](#) on the PYNQ-Z1 platform. This schedule represents a static allocation of memory operations and control states synchronized with the [FSM](#) of the [MNGR](#). Refer to [Fig. 3.2](#) for the dataflow graph. Although originally developed for the PYNQ-Z1, the same scheduling structure applies to bigger platforms. Only memory offsets and bank size will need to scale if the systolic size changes. Memory offsets define the address offset within a memory bank where data can be loaded or stored. With greater on-chip resource availability, larger operand tiles and tighter bank packing are achievable with reduced offset gaps and enhanced bandwidth. The schedule organizes execution into a series of [FSM](#) states, denoted as A, B, C, and so on, each corresponding to a specific memory phase or computational action. The memory banks labeled Mem A, Mem B, Mem D, and Mem Spare serve as the primary operand and output storage arrays. At any point in time, based on the schedule allocation, Mem A, Mem B, and Mem D can hold input activations, weights, intermediate results, or outputs, while Mem Spare provides overflow capacity for cases where operands cannot be cleanly partitioned.

Additionally, auxiliary [BRAM](#) arrays labeled Requant U1, U2, and U3 store parameters vectors—such as layer normalization biases, weight multipliers, or exponents, required for [NL](#) operations like [GELU](#), [Softmax](#), or [LayerNorm](#). We will refer to U1 to U3 as [S2V](#). The schedule defines a cyclical execution pattern in which a core sequence of [FSM](#) states is repeated across multiple phases to process each segment of the model. Each row in

the schedule maps a memory write or read operation to a specific address range, memory port, and operand function, such as loading matrix B, writing to D, or injecting [Softmax](#) outputs.

The primary strength of the scheduler lies in its flexibility and scalability. The [FSM](#) is explicitly defined with deterministic entry and exit conditions, governed by the state of the on-chip memory. This structure allows new computational blocks to be seamlessly integrated into the overlay with minimal changes. Once the operand starting addresses, access styles, and data bit-widths are specified, incorporating a new block requires only the addition of a corresponding [FSM](#) state with the appropriate control signal assignments.

Listing 4.1: Adding Softmax support to the FSM pseudocode

```
SOFTMAX: begin
    // ----- memA -----
    memA.wr_mode    <= 2'b00;    // ROW-WISE style write
    memA.rd_mode    <= 2'b10;    // AXI-style read
    memA.wr_offset  <= 0;        // Softmax output address = 0x0
    memA.rd_offset  <= 'h180;    // Softmax input address = 0x180
    // ----- memB -----
    memB.wr_mode    <= 2'b11;    // Abort writing
    memB.rd_mode    <= 2'b11;    // Abort reading
    memB.wr_offset  <= 'h0;
    memB.rd_offset  <= 'h0;
    // ----- memD -----
    memD.wr_mode    <= 2'b11;    // Abort writing
    memD.rd_mode    <= 2'b11;    // Abort reading
    memD.wr_offset  <= 'h0;
    memD.rd_offset  <= 'h0;
    // ----- S2V mems -----
    s2v.wr_en       <= 3'b000;    // Abort writing
    s2v.rd_en       <= 3'b000;    // Abort reading
    // ----- Precision -----
    memA.wr_x8_en   <= 1'b1;    // memA write precision = 8-bit
    memA.rd_x32_en  <= 1'b1;    // memA read precision = 32-bit
    // ----- Exit -----
    if (write_done_A) next_op <= NEXT_OP_IN_SEQ;
end
```

For instance, Listing 4.1 illustrates how support for a [Softmax](#) operation can be added

to the overlay, assuming it is a new block, the inputs are prefetched through one [AXI](#) stream, and the outputs are stored on-chip. After connecting the [Softmax](#) module [AXI](#) I/O interface to the [MNGR](#) I/O switch and registering its I/O dimensions through the dimension switch interface, the only modification necessary is the insertion of a new scheduler state. Assuming the [Softmax](#) input is 32-bit precision and stored in memory bank A at address 0x180, and the output is 8-bit precision to be written back to memory bank A at address 0x0, the new state within the [FSM](#) would activate the appropriate read/write paths and operand control lines accordingly. The [MNGR](#) core, discussed later in this chapter, handles these configurations and utilizes the partitioned [BRAM/URAM](#) blocks. The configuration, shown in Listing 4.1, primarily defines the read and write styles for memory A, memory B, memory D, and the [S2V](#) blocks. Since there is no [MM](#) multiplication in the added module, there is no configuration for multiplication operations, although support for them is available. Users must be aware of the functionality of the added block prior to enabling its configuration hooks.

In the case of [Softmax](#), the auxiliary parameters are too small to justify [S2V](#) allocation, so they are internally stored using a few [LUTs](#) inside the operation module itself. As a result, the [S2V](#) blocks are disabled for the added [FSM](#) state. Additionally, because [Softmax](#) expects its input to be streamed in an [AXI](#)-compatible manner, memory A’s read mode is set to [AXI](#)-style. The output of the [Softmax](#) module is written back element-wise in a row-wise fashion, so memory A’s write mode is set accordingly. Memory B and memory D are unused in this operation and are thus disabled.

The exit conditions for all [FSM](#) states are internally designed to be sticky. In this specific state, where only one coarse operation is executed, the ‘write\_done\_A’ one-pulse signal flags the end of execution once data is stable on-chip. However, to maintain dataflow integrity and avoid erratic synchronization—particularly when multiple coarse operations

are running in parallel—exit condition flags are latched until the transition to the next **FSM** state. Upon transition, all latches are cleared to indicate the start of a new execution phase.

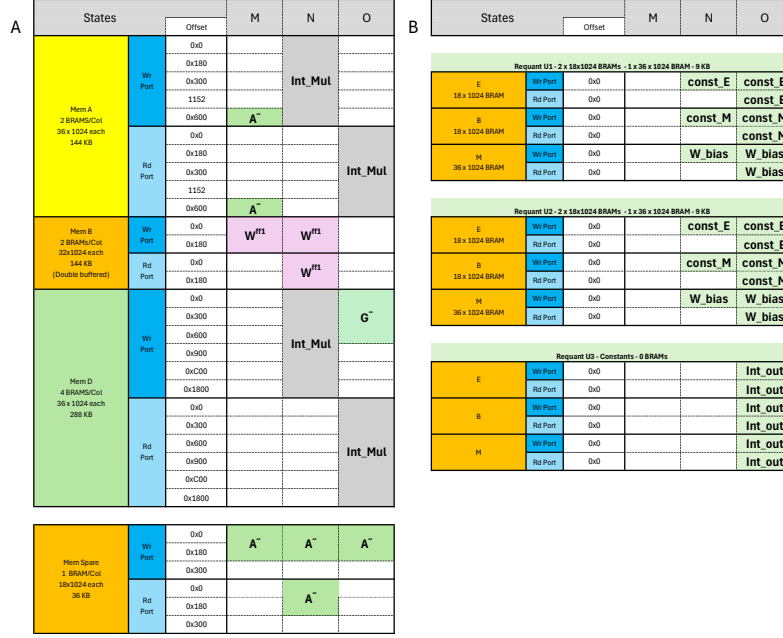


Figure 4.5: Memory access schedule for the IBERT intermediate sub-layer. A) Mem B loads weights  $W_{ff1}$ , Mem D accumulates intermediate results, and activations  $\tilde{A}$  are initially stored in Mem A before being moved to Mem Spare in state M to save space for overflow. Sub-layer output will be streamed at the state after O from Mem D. B) Constants and biases are stored at state N, and loaded at state O. U1 vectors overflow to U2. U3 is storing constants.

Another example demonstrating the capabilities of the scheduler is shown in Fig. 4.5, which illustrates the memory access schedule for the intermediate sublayer in the IBERT accelerator. The sub-layer comprises the largest **MM** multiplication operation in the model,

three requantization (NL) operations, and one GELU (NL) operation. Due to the large I/O size of this state, some of the streamed data exceeded the initially assigned address space. For example, although the MM multiplication destination memory block was D, the data overflowed into memory block A. Similarly, when the destination for S2V was U1, the data spilled into U2. These overflow conditions are managed internally.

## 4.2.2 Operands Routing and Runtime Dimension Configuration

Architecturally, the design comprises three main crossbar switches: The dimension switch, the main memory I/O switch, and the streaming-vector I/O switch. Each of the three crossbars has a specific function.

### Dimension Switching

The dimension switch is a register-based configuration unit that controls memory and dataflow dimensions during runtime. Rather than hardcoding tile sizes or buffer depths, this module receives runtime parameters such as matrix dimensions (M1, M2, M3), double buffer block counts, number of AXI streams, and memory partitioning factors (e.g.,  $M1/N$ ,  $M3/N$ ) and redistributes them internally across the memory system. For example, the switch computes the effective row count per bank for operand A, adjusts buffer reuse factors for operand B, and configures output matrix tiling for D.

### Main Memory I/O Switching

The main memory I/O switch enables memory-level routing between different layer blocks. This switch maps incoming data streams to distinct BRAM banks or memory regions

based on the control signals from the main scheduler. It also routes the outputs of the **BRAM** banks to their respective destination module. This switch is essential in the **AXI** compatibility of the design. At a given time, all the operation blocks will send a **READY** signal to the memory. This switch is essential in selecting which block does the handshake transaction with the memory **VALID** signal. The switch has the ability to send **AXI** transactions to  $N$  operands at a given time, where  $N$  is the number of supported operations on the overlay, and it can receive up to  $M+2$  **AXI** streams. One for memory A, one for memory D, and  $M$  for memory B, where  $M$  is the number of **memory-mapped to stream (MM2S)** interfaces used. This switch also operates in conjunction with the dimension switch to segment **BRAM** banks logically by operation. For example, when three **NL** operations are processed in parallel, the two switches dynamically adjust address offsets and routing logic to re-use the same physical banks while preserving logical separation. This allows the overlay to maximize **BRAM** reuse without violating data dependencies.

### **Streaming-vector Switching**

The last crossbar is the streaming-vector switch. Similar to the previous two switches, it interfaces with the control **FSM** to adapt vector-memory I/O depending on the active compute phase. This module is essential when multiple **NL** parameters are applied in parallel.

Because all memory-dependent modules derive their timing and addressing from these configurations, these switches act as a centralized point of synchronization for flexibility.

### 4.2.3 Memory Bank Logic and Access Control

The overall **MNGR** operation is organized through the **MNGR** core. It implements the physical interface to the operand and output memory blocks. It provides read and write control for **BRAM** banks allocated to operands. It issues enable signals to address generators, toggles buffer states for ping-pong **BRAM** access, and manages the alignment of memory windows with systolic computation timing. It also coordinates **AXI** handshake signals and ensures bus-level synchronization between **BRAM** blocks and external interfaces. Additionally, it tracks the execution mode (read/write) and ensures that access styles are respected throughout the pipeline. Its flexible design allows for bandwidth adaptation at multiple granularities (x8 to x512), supporting both compact and high-throughput modes of operation. The following sections outline the memory bank layout, buffering strategy, address generation mechanisms, and control algorithms used to ensure efficient and synchronized memory access throughout the accelerator pipeline.

| Systolic Size | Group | Banks /PE | Bank Depth (bit) | Bank Width (bit) | Min. Offsets/Bank | Total Size (KB) |
|---------------|-------|-----------|------------------|------------------|-------------------|-----------------|
| 8×8           | A     | 4         | 1024             | 36               | 0x180             | 144             |
|               | B     | 1         | 4096             | 36–144           | 0x180             | 144–576         |
|               | D     | 8         | 1024             | 36               | 0x300             | 288             |
|               | Spare | 2         | 1024             | 18               | 0x180             | 36              |
| 16×16         | A     | 2         | 1024             | 36               | 0x180             | 144             |
|               | B     | 1         | 4096             | 36–144           | 0x180             | 288–1152        |
|               | D     | 4         | 1024             | 36               | 0x300             | 288             |
|               | Spare | 1         | 1024             | 18               | 0x180             | 36              |
| 32×32         | A     | 1         | 1024             | 36               | 0x080             | 144             |
|               | B     | 1         | 4096             | 36–144           | 0x080             | 576–2304        |
|               | D     | 2         | 1024             | 36               | 0x180             | 288             |
|               | Spare | 1         | 1024             | 18               | 0x080             | 72              |

Table 6: Memory Layouts and Offsets for the Memory Manager

## Memory Bank Organization and Operand Mapping

The goal of the design is to maximize the throughput of the systolic multiplication operation. Accordingly, the target was to maximize the size of the systolic array and connect each of the systolic frontier I/Os to one or more **BRAMs/URAMs**, configured by the control **FSMs**. This configuration allows continuous streaming to and from the **MM** multiplier. Since the multiplication operation depends on two operands and a result, and since the memory layout is built around the multiplier, the on-chip **BRAMs/URAMs** are mainly divided into three groups: A, B, and D. The depth of each of these groups is inversely proportional to the systolic size. As discussed earlier in this chapter, when the matrices being multiplied are larger than the systolic array, each memory bank must store more data. Similarly, memory offsets decrease as the systolic size increases. Table 6 demonstrates the memory layouts and offsets for the accelerator at different systolic array sizes.

| Group                                                                                                                                                                                                                                                                                             | Access Behavior    | Write Style | Read Style | Write Granularity       | Read Granularity       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------|-------------|------------|-------------------------|------------------------|
| A                                                                                                                                                                                                                                                                                                 | AXI row-wise       | A           | A_to_MM    | 8b or 32b <sup>1</sup>  | 8b <sup>3</sup>        |
|                                                                                                                                                                                                                                                                                                   | AXI column-wise    | A_t         | B_to_MM    | 8b or 32b <sup>1</sup>  | 8b <sup>3</sup>        |
|                                                                                                                                                                                                                                                                                                   | Systolic read-back | D           | -          | 32b                     | -                      |
|                                                                                                                                                                                                                                                                                                   | AXI read-back      | -           | A_as_D     | -                       | 8b or 32b <sup>1</sup> |
| B                                                                                                                                                                                                                                                                                                 | AXI row-wise       | B_t*        | A_to_MM*   | -                       | -                      |
|                                                                                                                                                                                                                                                                                                   | AXI column-wise    | B           | B_to_MM    | 8b to 512b <sup>2</sup> | 8b <sup>3</sup>        |
|                                                                                                                                                                                                                                                                                                   | Systolic read-back | -           | -          | -                       | -                      |
|                                                                                                                                                                                                                                                                                                   | AXI read-back      | -           | A_as_D     | -                       | -                      |
| D                                                                                                                                                                                                                                                                                                 | AXI row-wise       | D_as_AB     | A_to_MM    | 8b or 32b <sup>1</sup>  | 8b <sup>3</sup>        |
|                                                                                                                                                                                                                                                                                                   | AXI column-wise    | D_as_AB     | B_to_MM    | 8b or 32b <sup>1</sup>  | 8b <sup>3</sup>        |
|                                                                                                                                                                                                                                                                                                   | systolic read-back | WR_D        | -          | 32b                     | -                      |
|                                                                                                                                                                                                                                                                                                   | AXI read-back      | -           | RD_D       | 32b                     | 8b or 32b <sup>1</sup> |
| * Disabled for IBERT to save resources.<br><sup>1</sup> Different granularity based on the current operation precision.<br><sup>2</sup> Up to 128b Byte addressable banks for direct <b>DMA</b> to <b>URAM</b> write support.<br><sup>3</sup> Writing to the systolic array in 8b precision only. |                    |             |            |                         |                        |

Table 7: Memory Group Access Behavior and Logic Mapping

Since the groups' blocks are reused across computation stages, they must support various read and write styles. Depending on the computation phase, memory groups can be written and streamed as:

- **AXI** row-wise, meaning that data are streamed to the **MNGR** element-by-element in a row-major fashion.
- **AXI** column-wise, meaning that data are streamed to the **MNGR** element-by-element in a column-major fashion.
- **AXI** readback, streaming to the **PL** or other computation blocks using the valid/ready handshake.
- Systolic read-back for writing the output of the systolic array to the **MNGR**.

Table 7 demonstrates the read/write styles supported by each group. Table 8 demonstrates the interpretation behind every access style.

| Access Style                                                  | Function | Description                                                                  |
|---------------------------------------------------------------|----------|------------------------------------------------------------------------------|
| A                                                             | Write    | Element-by-element row-wise stream of the given matrix.                      |
| A_t                                                           | Write    | Element-by-element write as the transpose of Matrix A, if needed.            |
| A_to_MM                                                       | read     | Load the frontier of a given matrix into the systolic array as Matrix A.     |
| A_as_D                                                        | read     | Rearrange Matrix A to be row-major, and read it in an AXI-compatible stream. |
| B                                                             | Write    | Column-by-column cyclic stream with double buffering and slicing.            |
| B_t                                                           | Write    | Write as the transpose of Matrix B, if needed.                               |
| B_to_MM                                                       | read     | Load the given matrix into the systolic array as Matrix B.                   |
| WR_D                                                          | Write    | Systolic-style diagonal write-back to Matrix D.                              |
| D_as_AB                                                       | Write    | Write as Matrix A or B based on need.                                        |
| RD_D                                                          | read     | Read Matrix D in a shuffled, AXI4-compatible stream.                         |
| * Assuming Matrix-Matrix Multiplication is $D = A \times B$ . |          |                                                                              |

Table 8: Read/Write Behavior Interpretation

For simplicity, group B does not support **AXI**-style read, as it is primarily used to store large weight matrices through double buffering and multi-**AXI** stream support. More

details about the referred write and read address generators will be discussed later in this chapter.

#### 4.2.4 Hazard-Resilient Double Buffering Controller

The double buffering controller alternates write and read access between two memory blocks. The controller's FSM and corresponding state assignment table are shown in Fig. 4.6 and Table 9, respectively. As a critical overlay component, it is designed to safeguard the system against synchronization issues and potential data hazards.

##### Corner Cases and Protection Mechanisms

- **Inter-state data integrity protection:** If one block is full and the second block has started the last write pipeline while the first block has not yet been completely read, the FSM triggers preemptive transitions to stall AXI and prevent overflow into unready memory banks.
- **Read-before-write hazards:** The first read operation is not initiated until operand A is written. Thereafter, switching between buffered blocks proceeds continuously with intermediate states that prevent reading a block until corresponding writes have fully completed, ensuring only complete data is consumed.
- **AXI stalling in non-safe states:** The system stall signal is asserted during transitions that involve read-write conflicts or buffer handovers (e.g. states highlighted in red in Fig. 4.6), providing a clean backpressure mechanism to upstream data sources.
- **Read-after-write conflicts with delayed read:** Since there is only one systolic

array, if the write of one buffer completes before the read of the other buffer begins, the **FSM** waits until the read is done, guaranteeing safe read initiation.

- **Safe state recovery:** On reset, the FSM initializes to state **A**, configuring Block 0 for write and Block 1 as idle. However, if the system has an odd number of blocks and resumes without an **FSM** reset, it may start from state **G**, which follows the same control flow but with Block 1 set for write and Block 0 as idle. The structured design of the **FSM** ensures the controller remains deterministic and hazard-free regardless of the starting condition.

| States     | Block 0 |    | Block 1 |    | Stall |
|------------|---------|----|---------|----|-------|
|            | rd      | wr | rd      | wr |       |
| A, default | 0       | 1  | 0       | 0  | 0     |
| B, G       | 0       | 0  | 0       | 1  | 0     |
| C_0, F_1   | 0       | 0  | 0       | 1  | 1     |
| C          | 0       | 0  | 0       | 0  | 1     |
| I          | 0       | 0  | 1       | 0  | 1     |
| C_1        | 1       | 0  | 0       | 1  | 1     |
| D, F_0     | 1       | 0  | 0       | 1  | 0/1   |
| H          | 0       | 1  | 1       | 0  | 0     |
| I_0        | 0       | 1  | 1       | 0  | 1     |
| I_1        | 0       | 1  | 0       | 0  | 1     |
| F          | 1       | 0  | 0       | 0  | 1     |

Table 9: Double-buffer controller output assignments.

The double-buffering infrastructure is designed to support any number of blocks as long as the block’s width is larger than the width of the systolic array architecture, which allows mapping bigger models on devices with less constrained on-chip memory resources.

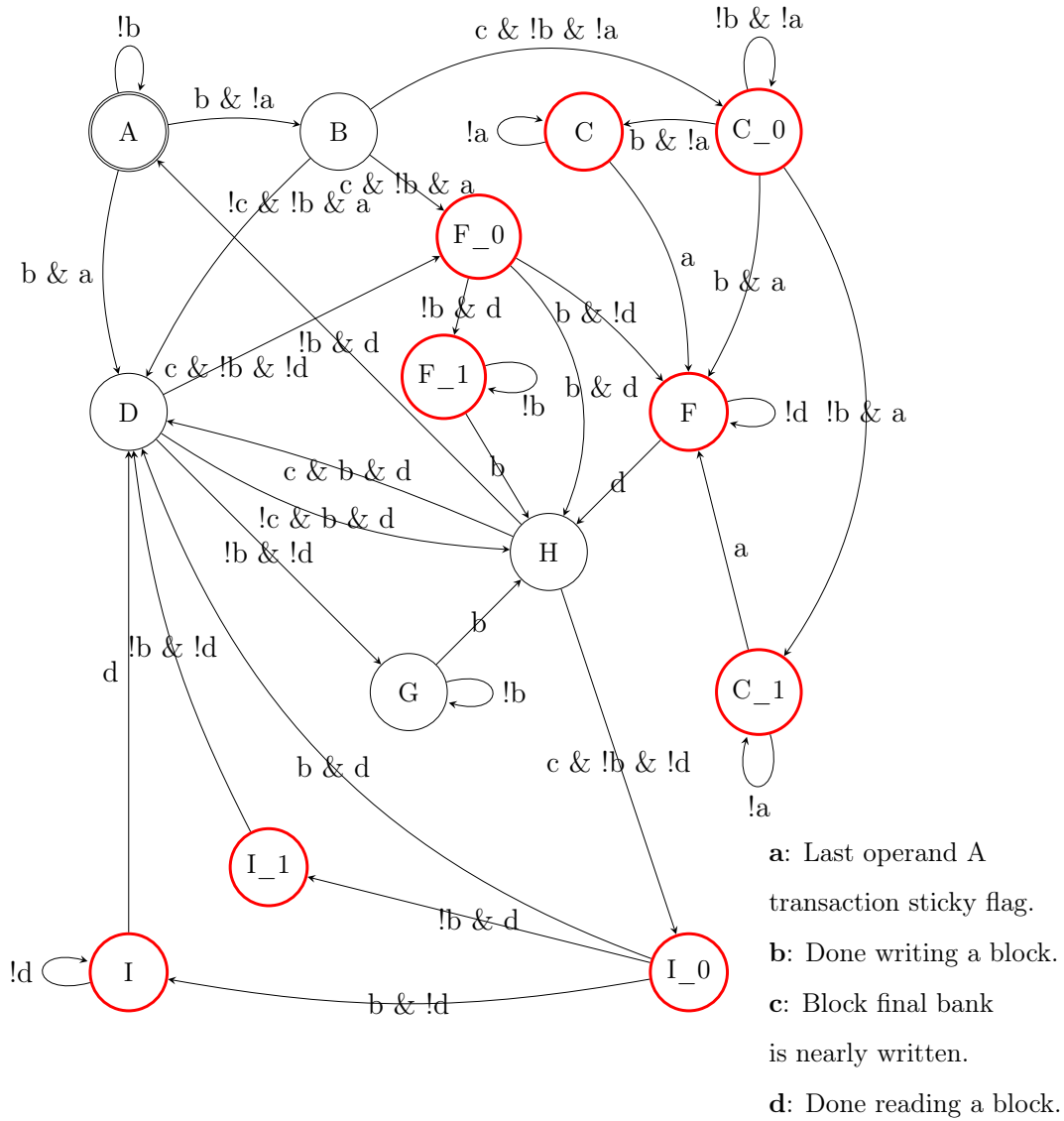


Figure 4.6: FSM for ping-pong memory control. Red-outlined states indicate AXI idle-hold conditions. Preemptive hazard conditions are detected in the `_0` states and subsequently captured in the corresponding base states (no suffix) if confirmed. Otherwise, the FSM transitions to the `_1` states, enabling controlled hazard release and resumption of normal operation.

## 4.2.5 Address Generation Algorithms

Dedicated address generation for each memory group are required to support flexible access patterns. These modules operate under control of the [MNGR FSM](#) and are parameterized based on the access style, packing factor, and base offset.

**Write Address Generation** Three classes of write address generators are supported:

- **AXI-Aligned Streaming:** Illustrated in Fig. 4.7, this mode incrementally generates addresses based on matrix depth ( $M_2$ ), systolic array depth ( $N$ ), and packing configuration. The operand matrix  $M_1 \times M_2$  is partitioned into  $N \times M_2$  sub-blocks and concatenated vertically across  $N$  memory banks. This format supports systolic injections and direct streaming to overlay modules or the CPU with minimal logic.

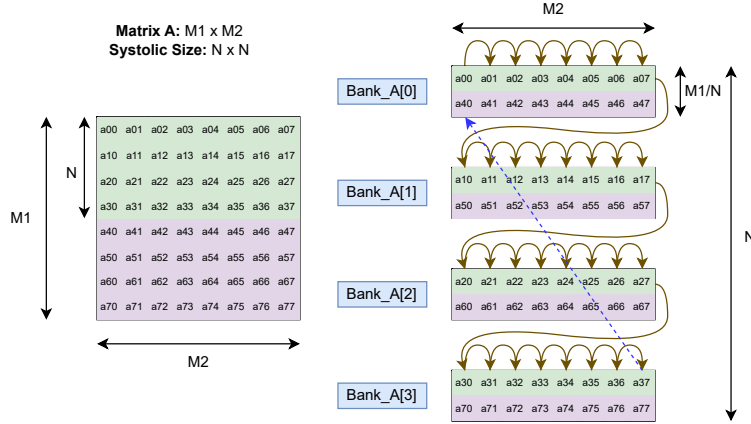


Figure 4.7: Illustration of AXI-Aligned Streaming. The matrix  $A \in \mathbb{R}^{M_1 \times M_2}$  is cyclically partitioned across  $N$  memory banks in phases of  $N$  rows each, resulting in a total of  $M_1/N$  phases. The brown lines show the row-wise stream and concatenation sequence of the elements within a phase. The dashed arrow marks the beginning of a new phase.

- Sliced Column-Major Streaming:** To support packed, cyclic, and column-major writes with double-buffered slicing, a specialized group of address generators was developed. These generators divide the input matrix into vertical slices aligned with the systolic array width ( $N$ ) and map them across multiple BRAMs/URAMs. The banks' selection is one-hot encoded to allow a straightforward column-major streaming. The packing logic ensures that data is aligned with the configured bus width and written in a format expected by the systolic read logic. The sliced data is concatenated vertically across the  $N$  banks. This alignment allows the double-buffered systolic multiplication algorithm described earlier to execute without additional multiplexing logic. Fig. 4.8 illustrates the address mapping algorithm.

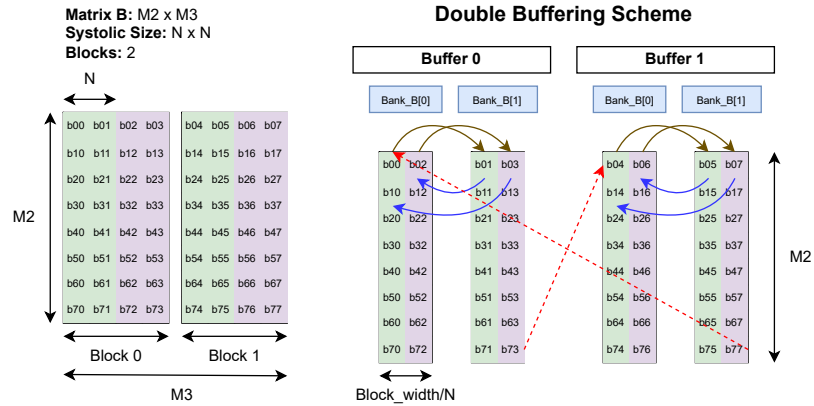


Figure 4.8: Double buffering write scheme for matrix  $B \in \mathbb{R}^{M_2 \times M_3}$ . The matrix is divided column-wise into blocks of arbitrary width. The double-buffer controller alternates writes between Buffer 0 and Buffer 1, each consisting of  $N$  BRAM banks. The data within a block is distributed cyclically across banks, resulting in  $\text{block\_width}/N$  phases per block. Brown arrows represent the write pattern within a phase, blue arrows indicate phase transitions, and red dashed arrows mark write buffer switches. When all buffers are full and additional blocks arrive, the system overwrites earlier blocks in a ping-pong fashion.

- Systolic Output Writeback:** As systolic output is shifted from the array's rightmost edge in phases, as explained earlier in this chapter, a specialized generator issue addresses that match this staggered timing. The write address generator accounts for this by issuing addresses that follow the systolic timing pattern, ensuring that output elements are committed to memory in the correct order for consistent output reuse in the overlay. Outputs are written contiguously into  $N$  banks, with packing disabled to preserve alignment and per-element precision. Unlike operand writes, all output banks remain active. The leftmost part of Fig. 4.9 shows the expected output from the systolic array, and the right part shows the reordered elements in the memory banks. This interleaved bank layout mirrors the AXI-aligned format, enabling seamless reuse by AXI-compliant modules without additional reshaping logic.

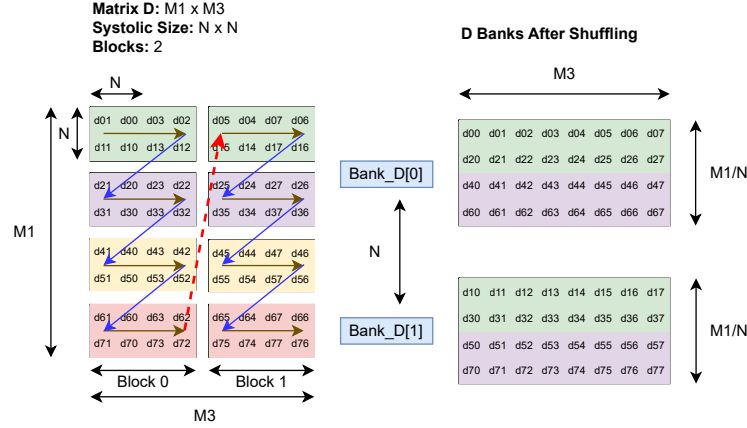


Figure 4.9: Systolic Output Writeback. Matrix  $D \in \mathbb{R}^{M_1 \times M_3}$  is written column-block by column-block. Each group of  $M_1/N$  rows forms a streaming phase, where output elements are written to the corresponding location across all banks (brown arrows). After completing one phase, the write address generator advances to the next set of rows (blue arrows), and after all phases in a block are written, the generator switches to the next block (red arrows).

**Read Address Generation** Read modes are aligned with the corresponding write strategies to preserve layout consistency and eliminate runtime reshaping:

- **AXI-Aligned Transfers:** To support external modules, vectorized post-processing, and CPU integration, a dedicated read path issues AXI-compatible data stream. It supports concatenated memory layouts, where sub-blocks of the matrix are serialized vertically across  $N$  banks. The generator walks linearly through these concatenated regions, issuing addresses based on the number of double-buffer blocks and systolic multiplier dimensions. Fig. 4.10 shows the address generation flow for this pattern.

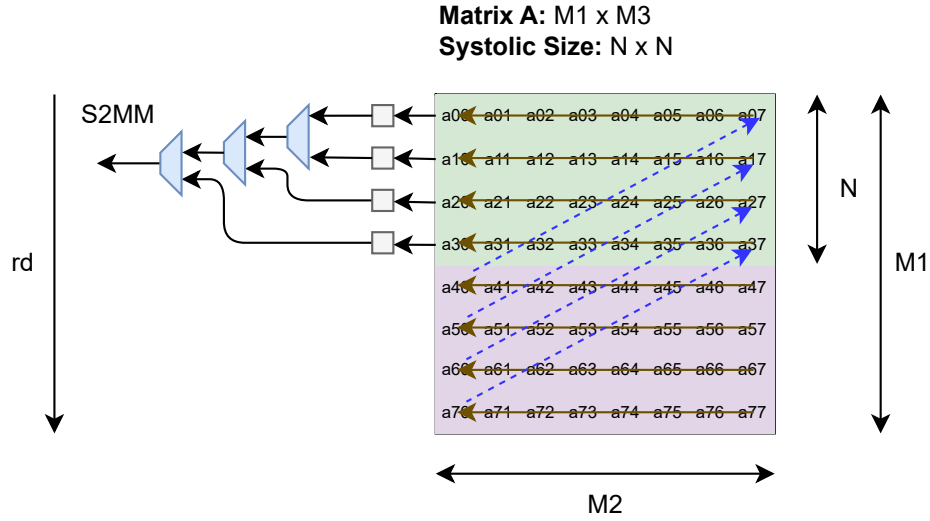


Figure 4.10: AXI-compatible streaming of matrix  $A \in \mathbb{R}^{M_1 \times M_3}$ . The matrix is divided into  $M_1/N$  streaming phases, each mapped to  $N$  consecutive rows. Within each phase, the elements are streamed sequentially as indicated by the brown lines. Dashed blue arrows mark the beginning of a new phase. This layout ensures compatibility with AXI bus constraints and enables direct injection into overlay modules without reshaping.

- **Row-wise and Column-wise Streaming:** Simple address counters are synchro-

nized with the systolic controller, iterating over row or column indices within active memory banks. The pre-aligned memory layout and write-address generators ensure data is directly streamable.

The address generators infrastructure supports the access behaviors detailed in Tables 7 and 8, enabling consistent operand streaming.

#### 4.2.6 Multi-AXI Stream Support

Multi-AXI streams refer to configurations with more than one PS-to-PL MM2S DDR interface. To achieve near-peak memory bandwidth utilization, the system must be capable of consuming all input streams at full rate, without excessive stalling. Neglecting this optimization results in a false hit of the memory wall during inference, where compute units remain idle waiting for data, even though the memory interface is underutilized. To address this, the MNGR architecture was designed to support multiple AXI streams, each with a maximum bus width of 128 bits.

Since the MM multiplication operation dominates compute time and the memory layout is built around the MM multiplication unit, it is logical to architect the multi-AXI streaming support around the systolic multiplier. This ensures maximal bandwidth utilization, as opposed to distributing AXI interfaces across heterogeneous operations or batching strategies. Integrating multi-AXI streaming directly into the core MM unit also enables scalability across devices with higher memory bandwidth and reusability for models with varying computational structures and batch sizes.

The infrastructure operates by dividing the memory banks connected to the MM multiplication unit’s input frontier into independent shreds. Each shred is assigned to a distinct

MM2S AXI system-level interface. Memory contiguity is preserved by mapping contiguous banks to the same shred—for example, with four AXI streams, the memory banks for group B are partitioned into four equal contiguous regions, assigned to shreds 0 through 3.

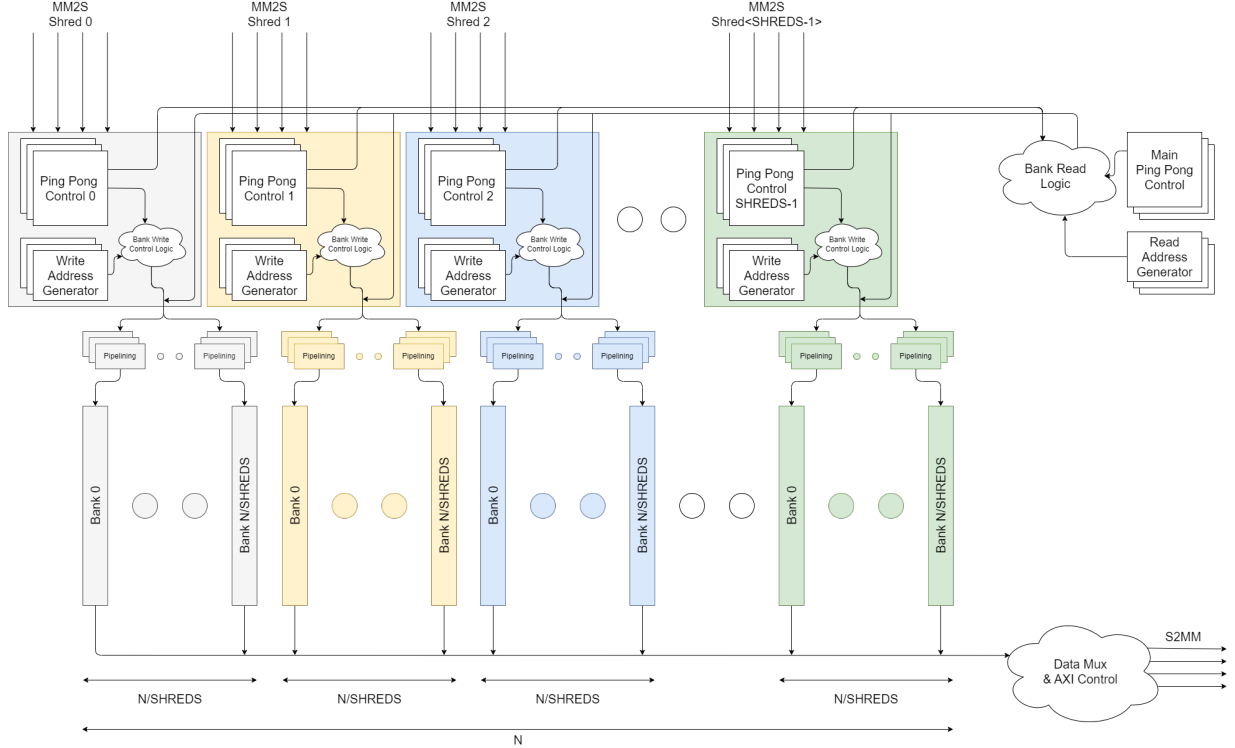


Figure 4.11: Multi-AXI memory manager architecture. The memory space connected to the MM multiplication unit is partitioned into multiple shreds, each independently handled by a separate MM2S AXI interface. Every shred includes a dedicated Ping Pong Controller, Write Address Generator, and bank write control logic. These components operate fully independently during the write phase. Once all shreds finish writing, a globally synchronized read phase is triggered, coordinated by the Bank Read Logic and Main Ping Pong Controller.

To mitigate the need for complex synchronization overhead, each shred’s write path operates independently via its own [AXI](#) interface, double-buffer controller, and address generators. This ensures full parallelism across shreds during the write phase from the [DDR](#) to the on-chip memory banks. However, reads are globally synchronized: once all double-buffer controllers signal the end of writing, a synchronized read operation is initiated across all shreds. This synchronization point ensures that all shreds reach a consistent state before any of them proceed, which is essential for race-free operation and data integrity. This coordinated read phase guarantees that multiple shreds can access and update shared memory regions through double buffering without data corruption. A shred is permitted to modify a memory region only after the synchronized read completes, at which point the previous data in memory is considered invalid. Fig. 4.11 illustrates the control scheme for shredding a wide matrix using multiple AXI streams.

#### 4.2.7 Streaming and Management of NL Vectors

The memory layout described earlier in this chapter has a maximum of three memory groups. To avoid unnecessary stalling, we assumed the system is capable of performing a maximum of three [NL](#) operations at a time. [NL](#) operations usually require auxiliary vectors, like biases, multipliers, exponents, constants, etc. Hence, an [S2V](#) unit was designed to complement the [MNGR](#) in scheduling the required resources for vector-based operations. Similar to the [MNGR](#), the [S2V](#) unit supports multi-grain addressing and operand overflow between memory banks within a sub-unit. The operands can overflow from "B" banks to "M" banks without stalling for large vectors. The memory layout of the [S2V](#) unit core is shown in Table 10. As shown, the core has three sub-units to accommodate the maximum throughout requirements highlighted earlier. The distinction between E, B, and M sub-

units reflects how different vector types are stored and streamed efficiently. Exponents (E) are typically smaller, low-precision vectors. Biases (B) are moderate-sized vectors. Multipliers (M) are often wider and higher in precision. The write operation to each sub-unit is asynchronous; however, the read is synchronous across the three sub-units within a unit.

| Unit    | Memory Sub-unit | Banks | Bank Depth (bit) | Bank Width (bit) | Total Size (KB) |
|---------|-----------------|-------|------------------|------------------|-----------------|
| U1 & U2 | E               | 1*    | 1024             | 18               | 2.25*           |
|         | B               | 1*    | 1024             | 18               | 2.25*           |
|         | M               | 1*    | 1024             | 36               | 4.5*            |
| U3      | E               | –     | Variable         | Variable         | <2.25           |
|         | B               | –     | Variable         | Variable         | <2.25           |
|         | M               | –     | Variable         | Variable         | <2.25           |

Table 10: Memory Layout for the S2V Unit. U1 & U2 are dedicated to large non-linear vectors, U3 is dedicated to constants and smaller vectors using LUT-based RAMs. Exponent (E), Multiplier (M), and Bias (B) are the three memory groups available in each S2V unit.\* The number of banks is a function of the target model’s hidden size.

## 4.2.8 Auxiliary Functions

In addition to supporting encoder-based inference, the resource subsystem includes auxiliary functions that can enable extensibility to decoder-based [LLMs](#). Decoder models require frequent reuse of intermediate outputs and more complex scheduling of matrix-matrix operations with looped dependencies. To support this, the memory manager includes row-wise output concatenation for efficient buffering of sequential outputs, matrix transpose operations for aligning token representations, and memory group interface switching (e.g., reading a matrix stored as D and treating it as A in the next timestep). Moreover, overflow handling from D into A simplifies data reuse without off-chip transfer, enabling support for

looped attention and decoder step unrolling. Furthermore, support for adding new AXI-compliant non-linear units, like [ReLU](#), provides infrastructure to expand the accelerator’s capabilities with minimal control logic changes.

### 4.3 Requantization Unit

The [RU](#) is a simple block that is used frequently in integer-only accelerators to quantize intermediate [INT32](#) signals to [INT8](#). It does a uniform symmetric quantization scheme where a real number  $q_{\text{in}}$  is uniformly mapped to an integer value  $q_{\text{out}}$  according to this equation:

$$q_{\text{out}} = \text{round} \left( \frac{(q_{\text{in}} + b) \cdot M}{2^E} \right)$$

Since the memory layout has a maximum of three memory groups, we assumed the system is capable of performing a maximum of three requantization operations at a time. Hence, the [RU](#), which is resource-cheap, was replicated three times and interfaced with the [MNGR](#) and [S2V](#). The [RU](#) can be replaced with any other needed [NL](#), or quantization, block as long as it has an AXI-compliant interface.

### 4.4 Non-Linear Unit and Matrix-Addition

The [NLU](#) necessary for showcasing the design was imported from the baseline custom IBERT accelerator with minor changes to allow better mapping on the chosen deployment devices’ primitives. Similarly, [MATADD](#) is a simple element-wise [MM](#) adder that was imported from the baseline accelerator with minor modifications to control the matrix-matrix addition using the centralized [FSM](#) instead of using local control units.

As discussed in this chapter, incorporating additional operations will only require appropriate interfacing with the [MNGR](#) and adding the necessary control states in the schedule. In the next chapter, we outline the experimental settings used to demonstrate the performance of our proposed centralized accelerator and report the findings.

# Chapter 5

## Experimental Setup and Results

In this chapter, we report the experimental results to demonstrate the performance of our proposed accelerator across three axes: resource consumption, power efficiency, and runtime performance. We compare our architecture on two target [FPGAs](#) (PYNQ-Z1 and KV260) and benchmark it against existing accelerators and embedded processors.

### 5.1 Experimental settings

The proposed accelerator is coded in SystemVerilog HDL and synthesized by Vivado 2022.1. The target devices were PYNQ-Z1 and AMD KV260 [FPGAs](#). The specifications of these devices are highlighted in Table 5. The maximum achieved clock frequencies were 300 MHz and 100 MHz for our performance comparison on the KV260 and PYNQ-Z1, respectively. Power consumption estimates were also obtained from the reports generated by the Vivado Design Suite based on the post-implementation timing simulation. The target model weights and activations were quantized, loaded onto an SD card, and then streamed to the

DDR memory of the target platforms using AXI-HP ports.

| DMA<br>BW (GB/s) | Systolic<br>Size | Device         | Frequency<br>(MHz) | Software<br>Overhead (ms) | Sequence<br>Latency (ms) |
|------------------|------------------|----------------|--------------------|---------------------------|--------------------------|
| 0.3              | 8×8              | KV260          | 300                | 10.25                     | 296.44                   |
| 0.3              | 16×16            | KV260          | 300                | 10.95                     | 277.84                   |
| 0.3              | 32×32            | KV260          | 300                | 10.31                     | 273.36                   |
| 0.4              | 8×8              | PYNQ-Z1        | 100                | 2.78                      | 484.11                   |
| <b>0.4</b>       | <b>16×16</b>     | <b>PYNQ-Z1</b> | <b>100</b>         | <b>2.6</b>                | <b>268.15</b>            |
| 1.2              | 8×8              | KV260          | 300                | 2.78                      | 146.70                   |
| 1.2              | 16×16            | KV260          | 300                | 2.60                      | 81.26                    |
| 1.2              | 32×32            | KV260          | 300                | 2.59                      | 76.32                    |
| 2.4              | 8×8              | KV260          | 300                | 1.12                      | 142.41                   |
| 2.4              | 16×16            | KV260          | 300                | 1.14                      | 49.20                    |
| 2.4              | 32×32            | KV260          | 300                | 1.18                      | 44.28                    |
| 4.8              | 8×8              | KV260          | 300                | 0.207                     | 140.06                   |
| 4.8              | 16×16            | KV260          | 300                | 0.028                     | 44.04                    |
| 4.8              | 32×32            | KV260          | 300                | 0.478                     | 28.26                    |
| 9.6              | 8×8              | KV260          | 300                | 3.20                      | 142.50                   |
| 9.6              | 16×16            | KV260          | 300                | 3.82                      | 47.16                    |
| 9.6              | 32×32            | KV260          | 300                | 2.61                      | 23.36                    |
| 19.2             | 8×8              | KV260          | 300                | 4.31                      | 143.12                   |
| 19.2             | 16×16            | KV260          | 300                | 3.68                      | 46.68                    |
| <b>19.2</b>      | <b>32×32</b>     | <b>KV260</b>   | <b>300</b>         | <b>3.03</b>               | <b>22.8</b>              |

Table 11: Measured runtime and software overhead for different configurations. The configurations in bold represent the setups with the least inference latencies for each device.

## 5.2 Performance Evaluation

We evaluated the runtime of the IBERT model on our accelerator with different systolic sizes, DMA bandwidths, number of AXI shreds, and devices. Table 11 demonstrates the performance sweep across these parameters. The maximum operating frequency we were able to achieve was 300 MHz on the KV260 board and 100 MHz on the PYNQ-Z1 board, using techniques like double buffering, pipelining, and floorplanning to close timing at the highly congested design with a 32x32 systolic multiplier and full-bandwidth support.

DMA bandwidth is defined as the number of bytes per AXI transaction multiplied by the frequency of operation.

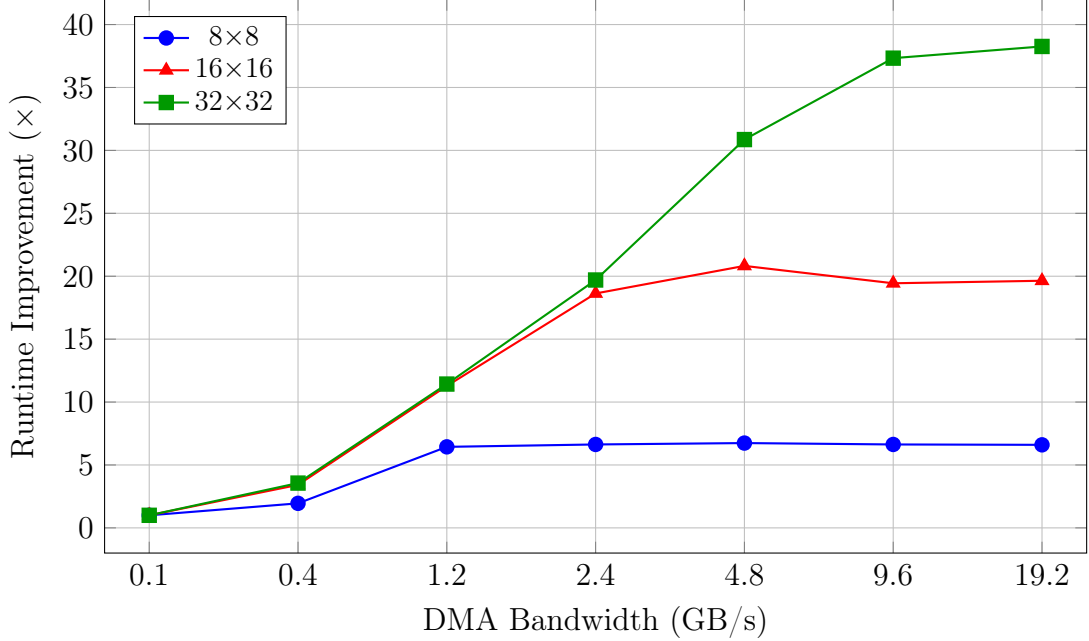


Figure 5.1: Inference runtime improvement factor relative to lowest-bandwidth (0.1 GB/s) baseline for systolic sizes  $8 \times 8$ ,  $16 \times 16$ , and  $32 \times 32$ . Higher is better.

As shown in Table 11, the operating frequency is fixed per device to get the accurate performance gain of changing the systolic array size. The operating frequencies was chosen as the lowest frequency that makes the biggest systolic array meet timing per device. As shown in Table 11, there is no significant improvement in running time with increasing the size of the systolic multiplier at low DMA bandwidths. Fig. 5.1 demonstrates the performance versus bandwidth behavior of the accelerator running on the  $8 \times 8$ ,  $16 \times 16$ , and  $32 \times 32$  systolic multipliers. For a given systolic size, the runtime saturates after a certain bandwidth. On the three tested sizes, there is no significant run-time improvement for

the 0.3 GB/s bandwidth. However, 32x32 and 16x16 systolic are having 1.92x and 1.8x improvement in runtime on the 1.2 GB/s bandwidth over the 8x8 configuration, respectively. Similarly, the 16x16 and 32x32 systolic multipliers have comparable performance at [DMA](#) bandwidths below 2.4 GB/s, with the 32x32 multiplier performing better with 1.56x improvement over the 16x16 configuration at 4.8 GB/s. After which the 16x16 configuration saturates. Beyond 9.6 GB/s, there is no significant improvement on the 32x32 configuration. Suggesting that the 32x32 multiplier saturates at 9.6 GB/s, and the system has to be expanded to at least a 64x64 systolic multiplier to fully utilize the maximum available bandwidth of the KV260 board. Table 12 highlights the configuration with the lowest latency for each systolic size. It can be noted that the 32x32 configuration has lower throughput per [DSP](#) compared to the 16x16 configuration, which confirms our observation that the accelerator throughput after the elbow points is bottlenecked by the compute resources since no performance gain is noticed with increasing the [DDR](#) bandwidth.

| <b>DMA<br/>BW (GB/s)</b> | <b>Systolic<br/>Size</b> | <b>Runtime<br/>Latency (ms)</b> | <b>Throughput<br/>(Seq/s)</b> | <b>DSPs</b>  | <b>Thr./DSP<br/>(<math>\times 10^{-3}</math>)</b> |
|--------------------------|--------------------------|---------------------------------|-------------------------------|--------------|---------------------------------------------------|
| 4.8                      | 8×8                      | 140.06                          | 7.14                          | 108 (8.7%)   | 66.11                                             |
| 4.8                      | 16×16                    | 44.04                           | 22.70                         | 300 (24.0%)  | 75.66                                             |
| 19.2                     | 32×32                    | 22.80                           | 43.86                         | 1068 (85.6%) | 41.07                                             |

Table 12: Latency, throughput, and DSP-normalized efficiency for the top-performing systolic configurations. Higher throughput per DSP is better. Lower latency is better.

Bandwidth utilization is also reported as a metric for assessing the efficiency of different platforms, as it indicates how closely an accelerator can approach the theoretical inference speed of the deployment device. Bandwidth utilization is demonstrated in Table 13. The bandwidth utilization is calculated as the inference bandwidth divided by the peak [DMA](#) bandwidth. The inference bandwidth is calculated by multiplying the measured throughput by the number of bytes needed to perform a sequence of inference. Due to the coarse [DMA](#)

granularity, we can’t report this metric exactly at the point where the memory bandwidth isn’t the bottleneck; however, the bandwidth utilization is reported at the first points after the elbow points for each configuration.

| Design   | DMA<br>BW (GB/s) | Systolic<br>Size | Runtime<br>Latency (ms) | Throughput<br>(Seq/s) | Inference<br>BW (GB/s) | BW Utilization |
|----------|------------------|------------------|-------------------------|-----------------------|------------------------|----------------|
| Ours     | 1.2              | 8×8              | 146.70                  | 6.82                  | 0.67                   | 55.56%         |
| Ours     | 2.4              | 16×16            | 49.20                   | 20.32                 | 1.99                   | 82.90%         |
| Ours     | 9.6              | 32×32            | 23.36                   | 42.81                 | 4.19                   | 43.61%         |
| Ours     | 19.2             | 32×32            | 22.80                   | 43.86                 | 4.29                   | 22.34%         |
| Baseline | 4.8              | –                | 53.5                    | 18.9                  | 1.85                   | 38.59%         |

Table 13: Performance and bandwidth utilization under different DMA bandwidths after the elbow points compared to the baseline.

The proposed accelerator can achieve maximum bandwidth utilization of 82.9% at the 16x16 systolic configuration with 2.4 GB/s DMA bandwidth, which indicates its capability to consume most of the available bandwidth given compute resources. The bandwidth drops to 22.34% and the throughput per DSP drops by 1.65x for the 32x32 configuration if the latency is the target design objective. Regardless, the low-bandwidth utilization is expected for encoder-based workloads as they perform large MM operations over the full input sequence, shifting the limiting factor to arithmetic throughput. However, Fig. 5.1, Table 12, and Table 13 demonstrate that our proposed architecture is capable of depleting up to 82.9% of the available bandwidth with the given compute-resources compared to the 38.5% maximum bandwidth utilization of the baseline at 4.8 GBs DMA bandwidth.

### 5.2.1 Bandwidth Constraints in Matrix Operations

To better understand the effect of the size of the multiplier and the input streams’ bandwidth on the performance saturation of our accelerator, we deployed the MM multiplication

operation in isolation. We measured its performance in comparison to the total latency of the model. As shown in Table 14, the IBERT-base model has around 228 MOPs/layer, totaling 2.74 GOPs for the model. The breakdown of runtime latencies with different systolic sizes and input bandwidth is shown in Fig. 5.2.

| Model        | Layers | Embed. Size | Attn. Heads | FFN Hidden Size | Parameters | Total OPs (GOPs) |
|--------------|--------|-------------|-------------|-----------------|------------|------------------|
| I-BERT-base  | 12     | 768         | 12          | 3072            | 125M       | 2.74             |
| I-BERT-large | 24     | 1024        | 16          | 3072            | 355M       | 3.45             |

Table 14: Dimensions of I-BERT [31] models at sequence length of 32.

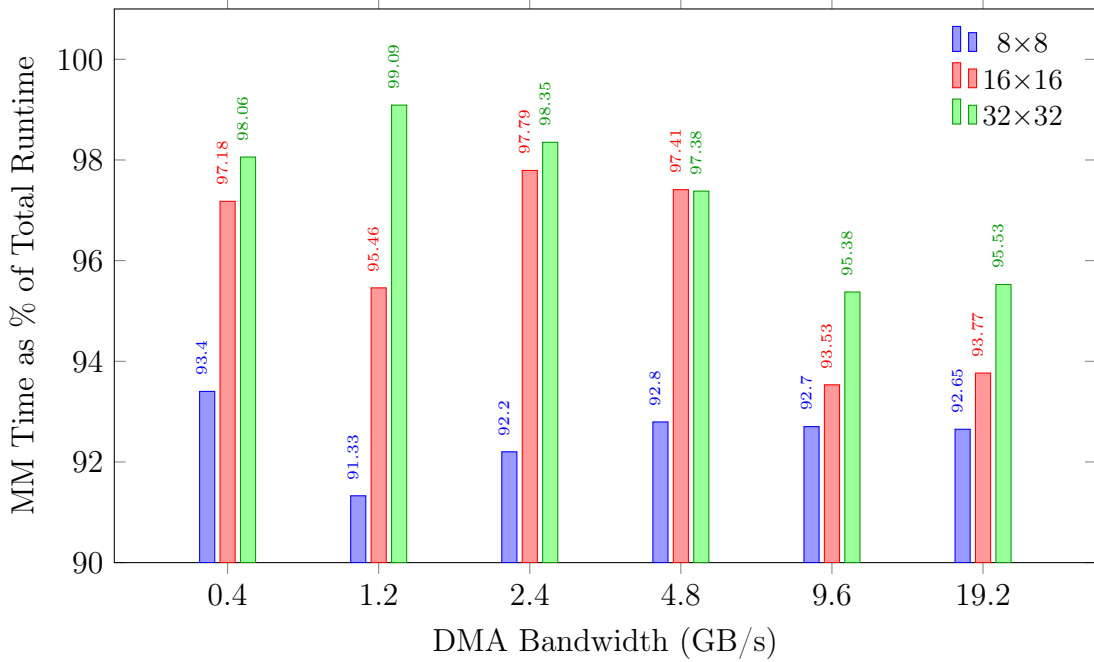


Figure 5.2: Percentage of matrix-matrix multiplication time relative to total inference runtime across different bandwidths and systolic array sizes.

At very low DMA bandwidth, high software overhead dilutes the multiplier dominance in the runtime latency. At low DMA bandwidth, MM multiplications dominate runtime,

indicating that **DMA** is the bottleneck. At the saturation point for each configuration, **MM** multiplication proportion drops as the total latency decreases, and the non-**MM** components’ effect on the latency becomes visible. The **MM** multiplication starts to saturate at high **DMA** bandwidth, indicating compute-bounding at 1.2 GB/s, 2.4 GB/s, and 9.6 GB/s for the 8x8, 16x16, and 32x32 configurations, respectively. After the saturation points, the **MM** time percentage drops as the **DMA** bandwidth increases because the **MM** performance gain doesn’t increase, and the software overhead required for buffer allocation and **DMA** setup increases. Hence, the **MM** dominance is slightly diluted.

The underutilization of bandwidth is happening due to interface stalling during the **MM** multiplications. Fig. 5.2 shows that the **MM** multiplications are consuming 91.3-99.4% of the accelerator runtime latency, which is close to the expected 97-99% range reported by the literature [56]. This indicates that the proposed resources-aware architecture is fully consuming the incoming stream, and the only way to improve is to move to a bigger device.

### 5.3 Resource Consumption Analysis

The resource consumption of the configurations at the elbow points of Fig. 5.1 is shown in Table 15. The on-chip memory layout is implemented by **BRAMs/URAMs**, the **MM** systolic multiplier, and the linear functions are implemented in **DSPs**, and the non-linear functions in the **NLU**, i.e., the **GELU** function, the division, square root, exponential operations, and **LayerNorm** functions, are approximated to be integer-only and implemented by **LUTs**. The breakdown of resources of each unit of the accelerator with the lowest inference latency on the KV260 is shown in Table 16.

As shown in Table 15, the centralized **MNGR** adds minor logic and interconnect utilization overhead compared to the baseline. This is expected as the centralized approach

| Design   | DMA<br>BW (GB/s) | Systolic<br>Size | AXI<br>DMAs | Device  | LUTs           | CLBs           | FFs            | DSPs          | BRAMs          | URAMs     |
|----------|------------------|------------------|-------------|---------|----------------|----------------|----------------|---------------|----------------|-----------|
| Ours     | 1.2              | 8×8              | 1           | PYNQ-Z1 | 20451 (38.44%) | 8263 (62.13%)  | 21542 (20.25%) | 108 (49.09%)  | 138 (98.57%)   | –         |
| Ours     | 2.4              | 16×16            | 4           | KV260   | 27568 (23.53%) | 6615 (45.18%)  | 42228 (18.03%) | 300 (24.04%)  | 108.5 (75.35%) | 16 (25%)  |
| Ours     | 4.8              | 32×32            | 1           | KV260   | 53360 (45.56%) | 13784 (94.15%) | 118764 (50.7%) | 1068 (85.58%) | 117.5 (81.6%)  | 64 (100%) |
| Ours     | 19.2             | 32×32            | 4           | KV260   | 63741 (54.42%) | 14406 (98.40%) | 134223 (57.3%) | 1068 (85.58%) | 133 (92.36%)   | 64 (100%) |
| Baseline | 4.8              | –                | 1           | KV260   | 63946 (54.6%)  | 14268 (97.5%)  | 125998 (53.8%) | 1123 (90.0%)  | 97.5 (68.1%)   | 64 (100%) |

Table 15: FPGA resource utilization for elbow points configurations and IBERT<sub>baseline</sub>. Percentages show usage relative to available hardware resources.

| Instance            | LUTs          | FFs           | BRAMs       | URAMs     | DSPs         |
|---------------------|---------------|---------------|-------------|-----------|--------------|
| MNGR                | 30100 (25.7%) | 29097 (12.4%) | 114 (79.2%) | 64 (100%) | 0 (0%)       |
| Systolic Multiplier | 17941 (15.3%) | 83050 (35.5%) | 0 (0%)      | 0 (0%)    | 1024 (82.1%) |
| RU                  | 700 (<1%)     | 423 (<1%)     | 0 (0%)      | 0 (0%)    | 12 (%)       |
| MATADD              | 46 (<1%)      | 38 (<1%)      | 0 (0%)      | 0 (0%)    | 0 (<1%)      |
| NLU                 | 4535 (3.8%)   | 4629 (2.0%)   | 0 (0%)      | 0 (0%)    | 32 (2.6%)    |
| Others              | 10419 (8.9%)  | 16986 (7.3%)  | 19 (13.2%)  | 0 (0%)    | 0 (0%)       |

Table 16: Resource breakdown per module for the 32×32 configuration at 19.2 GB/s, corresponding to the lowest-latency and highest-bandwidth setup.

requires heavy pipelining logic to close timing. Moreover, minor resource utilization is recorded when the bandwidth scales from 1 to 4 AXI interfaces. This overhead is mainly composed of 8.86% LUT and 6.6% FF increment, because of the multiplexing and pipelining logic introduced, and 10.76% BRAM increment due to DMA IP allocation.

## 5.4 Power Analysis and Comparison on Different Platforms

To assess the performance of our accelerator, we compare it against the embedded ARM Cortex-A9 processor available on KV260, as well as GPU-based inference. Table 17 shows the hardware specifications of the evaluated GPU platforms.

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

Table 17: Hardware specifications of the evaluated GPUs.

In Table 18, the power consumption and inference latency of the evaluated platforms are shown. The power results were adopted from the baseline design [4]. Our FPGA-based design achieves the lowest inference latency in the evaluated edge platforms. Showing at least a 23.63x improvement over CPU and a 1.17x improvement over the Jetson Orin Nano GPU. While higher-end GPUs offer higher raw throughput, our overlay is better suited for edge deployment.

| Device                              | Type | Power (W) | Latency (ms) |
|-------------------------------------|------|-----------|--------------|
| Apple M4                            | GPU  | 30        | 10.4         |
| Jetson Orin Nano (Low Power)        | GPU  | 9         | 38.4         |
| Jetson Orin Nano (High Performance) | GPU  | 11        | 26.7         |
| Ours                                | FPGA | 13        | 22.8         |

Table 18: Power consumption and latency comparison with embedded GPUs.

## 5.5 Comparison with Existing FPGA Research

Performance comparison between our proposed accelerator and existing research is shown in Table 19. Comparisons with research utilizing large cloud FPGAs with HBM [23] [59] [2] [9] [24] [30] can be hard due to the significant bandwidth disparity (460GB/s for Alveo U280 vs. 19.2GB/s for KV260). Nevertheless, we present the resource utilization for these works in Table 2, and we introduce the throughput per DSP as a performance metric in Table 19 to make the comparison with edge devices fair.

As shown in Table 19, our design does not achieve the lowest inference latency among

| Framework                 | Device     | Task         | Freq.<br>(MHz) | Power<br>(W) | Latency<br>(ms) | Thr.<br>(seq./s) | Thr./DSP<br>( $\times 10^{-3}$ ) | Energy Eff.<br>(seq./J) |
|---------------------------|------------|--------------|----------------|--------------|-----------------|------------------|----------------------------------|-------------------------|
| DFX [23]                  | Alveo U280 | GPT2-1.5B    | 200            | 45           | 47.6            | 21*              | 5.9                              | 0.466                   |
| FlightLLM [59]            | Alveo U280 | LLaMA2-7B    | 225            | 45           | 18.2            | 55*              | 8.7                              | 1.22                    |
| EdgeLLM [24]              | Alveo U280 | ChatGLM-6B   | 250            | 50.7         | 13.3            | 75*              | 13.4                             | 1.47                    |
| LlamaF [56]               | ZCU102     | TinyLLaMA    | 205            | 5.08         | 666.7           | 1.5*             | 2.84                             | 0.29                    |
| Li et al [34]             | KV260      | LLaMA2-7B    | 300            | 6.57         | 204.0           | 4.9*             | 16.8                             | 0.74                    |
| FET-OPU [3]               | Alveo U200 | RoBERTa-base | 200            | 7.4          | 6.82            | 146.6            | 30.1                             | 19.8                    |
| FTRANS [33]               | VCU118     | RoBERTa-base | 170            | 25.1         | 10.6            | 94.2             | 14.4                             | 3.75                    |
| NPE [30]                  | Z-7100     | BERT-base    | 200            | 20           | 7.40            | 135.1            | 66.9                             | 6.75                    |
| VITA [44]                 | ZC7020     | ViT-base     | 150            | 0.9          | 363.64          | 2.7              | 12.4                             | 3                       |
| PP-Transformer [8]        | U200       | BERT-base    | 200            | –            | 1.27            | 787.4            | 0.11                             | –                       |
| IBERT <sub>baseline</sub> | KV260      | RoBERTa-base | 250            | 12           | 53.5            | 18.9             | 16.8                             | 1.57                    |
| Ours                      | KV260      | RoBERTa-base | 300            | 13           | 22.8            | 43.9             | 41.06                            | 3.37                    |

Table 19: Performance comparison across FPGA-based language models frameworks. Lower latency is better. Higher throughput (Thr.), energy efficiency (Energy Eff.), and throughput/DSP (Thr./DSP) are better. \* Decoder-based models’ throughput is tokens/s.

the compared frameworks, due to the difference in available resources. However, it demonstrates superior throughput efficiency per [DSP](#), proving its compatibility for edge-class [FPGAs](#) implementations. Although the NPE [\[30\]](#) framework reports a higher throughput per [DSP](#), our design achieves comparable efficiency at nearly half the power consumption. This indicates effective utilization of compute resources within a constrained power budget.

For energy efficiency, FET-OPU [\[3\]](#) and FTRANS [\[33\]](#) show significantly higher performance with higher energy efficiency; however, these two frameworks are deployed on cloud-class devices. The abundance in resources enabled more flexible implementations and less congestion on the fabric. Hence, higher energy efficiency and higher performance are achieved. On the other hand, FET-OPU [\[3\]](#) and FTRANS [\[33\]](#) suffer from 4.5x and 6.1x [DSP](#) utilization overhead, respectively, compared to our design.

### 5.5.1 Comparison with IBERT Baseline

As shown in Tables [15](#) and [19](#), although the centralized [MNGR](#) infrastructure adds minor logic and interconnect utilization overhead of 3.5% FFs and 0.9% CLBs. An overhead of 35.5% [BRAM](#) utilization is also observed; however, it is expected as the [MNGR](#) establishes a connection with the four [DMA](#) streams available on the KV260 board, compared to only one stream in the baseline. Based on the channel bandwidth, each established [DMA](#) link is allocated few on-chip memory resources for the [DMA](#) IP. 4. 42% reduction in [DSP](#) utilization is observed due to the single large systolic array (32x32) in our architecture, which consumes less [DSP](#) than the smaller systolic arrays in the combined baseline implementation.

When evaluated on the same model and device, the observed logic and interconnect utilization overhead is traded-off with 2.32x latency, 2.44x throughput per [DSP](#), and up

to 2.1x bandwidth utilization improvements. Moreover, a 2.14x improvement in energy efficiency was achieved. For edge applications, the minor increase in resources is acceptable.

# Chapter 6

## Conclusion

### 6.1 Conclusion

This thesis presented a resource-managed accelerator architecture for integer encoder-based transformer models deployment on edge-class [FPGAs](#). The core contributions of this work are:

- A double-buffered centralized [MNGR](#) that supports multi-stream [AXI DMA](#) multiplexing and adapts to varying memory bandwidth conditions through configurable buffer sizes and phased access patterns, without the need for back-end [DMA](#) multiplexing. The [MNGR](#) achieves an improvement of up to 2.1x bandwidth utilization over the baseline with minimal on-chip multiplexing and pipelining overhead.
- A resource-managed [FPGA](#) accelerator for integer-only encoder-based transformers inference. The dataflow of integer encoder-only transformer architectures is deployed

as a schedule of operations in this design to maximize on-chip data reuse and system throughput.

- Evaluation of the system using the IBERT [31] model on low-energy edge devices, achieving up to **2.32× latency speedup** and **2.14× energy efficiency improvement** over the IBERT<sub>baseline</sub> and competitive performance compared to GPU-class embedded accelerators, despite tighter power and resource constraints.

The system demonstrated up to **2.32× latency improvement** over the IBERT<sub>baseline</sub> design and **1.17× speedup** over the Nvidia Jetson Orin Nano GPU. At peak, the design achieved **82.9% bandwidth utilization**. Moreover, our results demonstrate that centralized memory management achieves high **DSP** utilization and efficient **BRAM** usage. Systolic array scaling was achieved up to the devices’ resource limits, consuming **91.3-99.4%** of the accelerator runtime, which is close to the 97-99% expected range. Compared to distributed memory designs, our approach reduces control complexity and achieves comparable power-delay tradeoffs, confirming that centralizing the on-chip memory resources with scheduling and pipelining can be a viable and effective **LLM** acceleration approach for resource-constrained FPGAs.

## 6.2 Limitations and Future Work

Despite achieving strong performance on edge-class **FPGAs**, directions for future work remain. Including:

- **Support for decoder-based models:** The system currently targets encoder-only architectures. Extending support to decoder or encoder-decoder models would re-

quire enhancements to the [MNGR](#), including KV caching, post-QKV positional encoding, and improved [DMA](#) strategies for fine-grained per-token communication.

- **Workload Expansion:** This work demonstrated the capabilities of the proposed architecture using the IBERT-base model. For a more comprehensive evaluation, the accelerator’s performance can be demonstrated on larger models. Most of the required effort would involve building the supporting software infrastructure for training, quantizing, and exporting model parameters. Then, the existing design can be adapted by modifying the schedule to align with the new dataflow graph, and interfacing any additional [AXI](#)-compliant [NL](#) blocks, if needed.
- **Scalability of interconnects:** The current use of crossbars for operand routing can introduce area and timing overhead as the system scales. For models with a large number of concurrent compute blocks, this might become inefficient. Future work should explore integrating lightweight NoC-based interconnects into our design to improve its scalability and modularity.

# References

- [1] Keivan Alizadeh, Seyed Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. LLM in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12562–12584, 2024.
- [2] Yueyin Bai, Hao Zhou, Keqing Zhao, Jianli Chen, Jun Yu, and Kun Wang. Transformer-OPU: An FPGA-based overlay processor for transformer networks. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 221–221, 2023.
- [3] Yueyin Bai, Hao Zhou, Keqing Zhao, Hongji Wang, Jianli Chen, Jun Yu, and Kun Wang. FET-OPU: A flexible and efficient FPGA-based overlay processor for transformer networks. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.
- [4] Marat Bekmyrza. Language Model Inference on FPGA with Integer-only Operations. 2025.

- [5] Peter Belcak and Roger Wattenhofer. Exponentially Faster Language Modelling. *arXiv preprint arXiv:2311.10770*, 2023.
- [6] Aishwarya Bhandare, Vamsi Sripathi, Deepthi Karkada, Vivek Menon, Sun Choi, Kushal Datta, and Vikram Saletore. Efficient 8-bit Quantization of Transformer Neural Machine Translation Model. *arXiv preprint arXiv:1906.00532*, 2019.
- [7] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. GPT-NEO: Large scale autoregressive language modeling with mesh-tensorflow. *URL: <https://doi.org/10.5281/zenodo.5297715>*, 2022.
- [8] Jialin Cao, Xuanda Lin, Manting Zhang, Kejia Shi, Jun Yu, and Kun Wang. PP-transformer: Enable efficient deployment of transformers through pattern pruning. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.
- [9] Hongzheng Chen, Jiahao Zhang, Yixiao Du, Shaojie Xiang, Zichao Yue, Niansong Zhang, Yaohui Cai, and Zhiru Zhang. Understanding the Potential of FPGA-based Spatial Acceleration for Large Language Model Inference. *ACM Transactions on Reconfigurable Technology and Systems*, 18(1):1–29, 2023.
- [10] Xiang Chen, Jindong Li, and Yong Zhao. Hardware Resource and Computational Density Efficient CNN Accelerator Design Based on FPGA. In *2021 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*, pages 204–205, 2021.
- [11] Jaewan Choi, Hailong Li, Byeongho Kim, Seunghwan Hwang, and Jung Ho Ahn. Accelerating transformer networks through recomposing softmax layers. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 92–103, 2022.

- [12] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. PACT: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.
- [13] Dimitrios Danopoulos, Georgios Zervakis, Dimitrios Soudris, and Jörg Henkel. Transaxx: Efficient transformers with approximate computing. *IEEE Transactions on Circuits and Systems for Artificial Intelligence*, 2025.
- [14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [15] Peiyan Dong, Mengshu Sun, Alec Lu, Yanyue Xie, Kenneth Liu, Zhenglun Kong, Xin Meng, Zhengang Li, Xue Lin, Zhenman Fang, et al. HeatVit: Hardware-efficient Adaptive Token Pruning for Vision Transformers. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 442–455, 2023.
- [16] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, and Houlsby Neil. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *arXiv preprint arXiv:2010.11929*, 2020.
- [17] Hongxiang Fan, Thomas Chau, Stylianos I Venieris, Royson Lee, Alexandros Kouris, Wayne Luk, Nicholas D Lane, and Mohamed S Abdelfattah. Adaptable butterfly accelerator for attention-based NNs via hardware and algorithm co-design. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 599–615, 2022.

- [18] Haoqiang Guo, Lu Peng, Jian Zhang, Qing Chen, and Travis D LeCompte. Att: A fault-tolerant RERAM accelerator for attention-based neural networks. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 213–221, 2020.
- [19] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-eye: A complete design flow for mapping CNN onto embedded FPGA. *IEEE transactions on computer-aided design of integrated circuits and systems*, 37(1):35–47, 2017.
- [20] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, and Deong-Kyoon Jeong. A<sup>3</sup>: Accelerating attention mechanisms in neural networks with approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 328–341, 2020.
- [21] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W Lee. ELSA: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 692–705, 2021.
- [22] Jude Haris, Rappy Saha, Wenhao Hu, and José Cano. Designing efficient LLM accelerators for edge devices. *arXiv preprint arXiv:2408.00462*, 2024.
- [23] Seongmin Hong, Seungjae Moon, Junsoo Kim, Sungjae Lee, Minsub Kim, Dongsoo Lee, and Joo-Young Kim. DFX: A Low-Latency Multi-FPGA Appliance for Accelerating Transformer-Based Text Generation. In *55th IEEE/ACM International Symposium on Microarchitecture*, pages 616–630, 2022.

- [24] Mingqiang Huang, Ao Shen, Kai Li, Haoxiang Peng, Boyu Li, and Hao Yu. EdgeLLM: A highly efficient CPU-FPGA heterogeneous edge accelerator for large language models. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2025.
- [25] Suyeon Hur, Seongmin Na, Dongup Kwon, Joonsung Kim, Andrew Boutros, Eriko Nurvitadhi, and Jangwoo Kim. A fast and flexible FPGA-based accelerator for natural language processing neural networks. *ACM Transactions on Architecture and Code Optimization*, 20(1):1–24, 2023.
- [26] Hanhwi Jang, Joonsung Kim, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. MnnFast: A fast and scalable system architecture for memory-augmented neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 250–263, 2019.
- [27] Christoforos Kachris. A survey on hardware accelerators for large language models. *Applied Sciences*, 15(2):586, 2025.
- [28] Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, Amir Yazdanbakhsh, and Tushar Krishna. Flat: An optimized dataflow for mitigating attention bottlenecks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 295–310, 2023.
- [29] Nitish Shirish Keskar, Bryan McCann, Lav R Varshney, Caiming Xiong, and Richard Socher. Q8BERT: Quantized BERT. *arXiv preprint arXiv:1909.05858*, 2019.
- [30] Hamza Khan, Asma Khan, Zainab Khan, Lun Bin Huang, Kun Wang, and Lei He. NPE: An FPGA-based overlay processor for natural language processing. *arXiv preprint arXiv:2104.06535*, 2021.

- [31] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. IBERT. *arXiv preprint arXiv:2101.01321v3*, 2021.
- [32] Ann Franchesca Laguna, Arman Kazemi, Michael Niemier, and X Sharon Hu. In-memory computing based accelerator for transformer networks for long sequences. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1839–1844, 2021.
- [33] 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*, pages 175–180, 2020.
- [34] 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. *arXiv preprint arXiv:2502.10659*, 2025.
- [35] Jindong Li, Guobin Shen, Dongcheng Zhao, Qian Zhang, and Yi Zeng. Firefly: A High-Throughput Hardware Accelerator for Spiking Neural Networks with Efficient DSP and Memory Optimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 31(8):1178–1191, 2023.
- [36] Zhengang Li, Mengshu Sun, Alec Lu, Haoyu Ma, Geng Yuan, Yanyue Xie, Hao Tang, Yanyu Li, Miriam Leeser, Lin-Xue Wang, Zhangyang, 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.

- [37] Zhikai Li and Qingyi Gu. I-vit: Integer-only quantization for efficient vision transformer inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 17065–17075, 2023.
- [38] Shiwei Liu, Tianlong Chen, Zhenyu Zhang, Xuxi Chen, Tianjin Huang, Ajay Jaiswal, and Zhangyang Wang. Sparsity may cry: Let us fail (current) sparse neural networks together! *arXiv preprint arXiv:2303.02141*, 2023.
- [39] 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. *arXiv preprint arXiv:1907.11692*, 2019.
- [40] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. LLaMA 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288*, 2023.
- [41] Zejian Liu, Gang Li, and Jian Cheng. Hardware Acceleration of Fully Quantized BERT for Efficient Natural Language Processing. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 513–517, 2021.
- [42] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 977–991, 2021.
- [43] Siyuan Lu, Meiqi Wang, Shuang Liang, Jun Lin, and Zhongfeng Wang. Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer. In *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*, pages 84–89, 2020.

- [44] Shashank Nag, Gourav Datta, Souvik Kundu, Nitin Chandrachoodan, and Peter A. Beerel. ViTA: A Vision Transformer Inference Accelerator for Edge Applications. In *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2023.
- [45] Azade Nova, Hanjun Dai, and Dale Schuurmans. Gradient-free structured pruning with unlabeled data. In *International Conference on Machine Learning*, pages 26326–26341, 2023.
- [46] Ikumi Okubo, Keisuke Sugiura, and Hiroki Matsutani. A Cost-Efficient FPGA-Based CNN-Transformer Using Neural ODE. *IEEE Access*, 2024.
- [47] Hongwu Peng, Shaoyi Huang, Tong Geng, Ang Li, Weiwen Jiang, Hang Liu, Shusen Wang, and Caiwen Ding. Accelerating transformer-based deep learning models on FPGAs using column balanced block pruning. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, pages 142–148, 2021.
- [48] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2018.
- [49] Shrihari Sridharan, Jacob R Stevens, Kaushik Roy, and Anand Raghunathan. X-former: In-memory acceleration of transformers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 31(8):1223–1233, 2023.
- [50] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*, 2023.
- [51] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, and Faisal

- Azhar. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
- [52] Georgios Tzanos, Christoforos Kachris, and Dimitrios Soudris. Hardware acceleration of transformer networks using FPGAs. In *2022 Panhellenic Conference on Electronics & Telecommunications (PACET)*, pages 1–5, 2022.
- [53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [54] Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 97–110, 2021.
- [55] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant. In *International Conference on Machine Learning*, pages 38087–38099, 2023.
- [56] Han Xu, Yutong Li, and Shihao Ji. LlamaF: An Efficient Llama2 Architecture Accelerator on Embedded FPGAs, 2024.
- [57] Nan Yang, Tao Ge, Liang Wang, Binxing Jiao, Daxin Jiang, Linjun Yang, Rangan Majumder, and Furu Wei. Inference with reference: Lossless acceleration of large language models. *arXiv preprint arXiv:2304.04487*, 2023.
- [58] Xiaoxuan Yang, Bonan Yan, Hai Li, and Yiran Chen. ReTransformer: ReRAM-based processing-in-memory architecture for transformer acceleration. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

- [59] Shulin Zeng, Jun Liu, Guohao Dai, Xinhao Yang, Tianyu Fu, Hongyi Wang, Shiyao Li, Wenheng Ma, Hanbo Sun, Zixiao Huang, Zehao Wang, Yadong Dai, Ruoyu Zhang, Jintao Li, Kairui Wen, Xuefei Ning, and Yu Wang. FlightLLM: Efficient Large Language Model Inference with a Complete Mapping Flow on FPGA. *arXiv preprint arXiv:2401.03868*, 2024.
- [60] Biao Zhang and Rico Sennrich. Root Mean Square Layer Normalization. In *Advances in Neural Information Processing Systems*, volume 32, 2019.
- [61] Zhe Zhou, Junlin Liu, Zhenyu Gu, and Guangyu Sun. Energon: Toward efficient acceleration of transformers using dynamic sparse attention. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(1):136–149, 2022.