Accelerating SmolVLA on an FPGA Using Allo

Sam	**Ezra**	**Stanley**	**Isabella**
Attention Layer	Attention Layer	MLP Layer	MLP Layer
srb343@cornell.edu	er495@cornell.edu	ss3679@cornell.edu	isf9@cornell.edu

Abstract

Vision-Language-Action (VLA) models represent a significant step towards general-purpose robot control, integrating visual perception and language understanding to synthesize complex actions. However, the deployment of such models on edge devices is hindered by their substantial computational and memory bandwidth requirements. This work presents an FPGA-based accelerator for **SmolVLA**, a compact VLA model tailored for efficient robotic control. We leverage **Allo**, a composable high-level synthesis language, to design and optimize key computational kernels, specifically focusing on the Self-Attention and Multi-Layer Perceptron (MLP) layers within the model's Vision Encoder component. By exploiting the spatial parallelism and reconfigurability of the Xilinx Alveo U280 FPGA, we implement efficient hardware structures including tiled matrix multiplications and systolic arrays. We provide a detailed analysis of the workload, describe our hardware implementation strategy using Allo, and evaluate the performance of our accelerator in terms of latency and resource utilization, demonstrating the feasibility and benefits of FPGA acceleration for edge-based VLA inference.

1. Introduction

TODO: Ezra Reiss **(50%)**

Project Context:

- Introduce the problem: Efficiently running VLA (Vision-Language-Action) models on edge devices.
- Mention "SmolVLA" as the specific target workload.
- State the thesis: FPGA acceleration using Allo.
- Outline the contributions:
  1. Analysis of SmolVLA computational requirements.
  2. Implementation of key kernels using Allo.
  3. Evaluation of performance/efficiency on U280.

Recent advances in Vision-Language-Action (VLA) models have demonstrated the efficacy of integrating visual perception and language understanding for control tasks. However, deploying these models on edge devices remains a significant challenge due to their substantial computational requirements and memory bandwidth constraints. This project focuses on accelerating **SmolVLA**, a compact VLA model designed for efficient robot control, using Field-Programmable Gate Arrays (FPGAs).

While General Purpose GPUs are the standard for training and inference in data centers, FPGAs offer a compelling alternative for edge robotics due to their low latency, deterministic execution, and high energy efficiency. Our work leverages **Allo**, a high-level accelerator design language developed at Cornell University, to implement and optimize the key computational kernels of SmolVLA on a Xilinx Alveo U280 FPGA.

We specifically target the Vision Encoder component of the SmolVLA architecture, responsible for generating visual embeddings from camera inputs. This report makes the following contributions:
1. A detailed analysis of the computational and memory demands of the SmolVLA Vision Encoder.

2. An implementation of the Self-Attention and Multi-Layer Perceptron (MLP) layers using Allo's composable optimizations.
3. An evaluation of the accelerator's performance in terms of latency and resource utilization (DSP, BRAM) on the U280 platform.

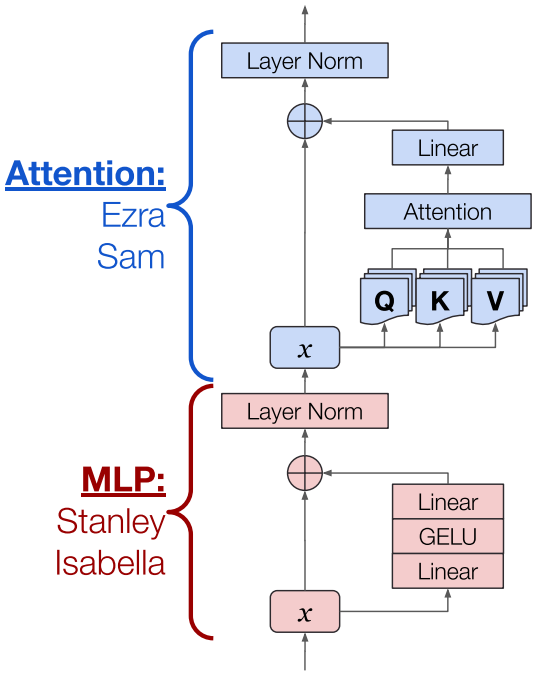

Fig. 1: **High-level architecture of the SmolVLA Transformer block.** The design is composed of two primary sub-modules: the Multi-Head Attention mechanism (top) and the Multi-Layer Perceptron (MLP) with GELU activation (bottom). The diagram illustrates the dataflow through the Linear projections, residual connections, and Layer Normalization steps that constitute a single layer.

2. Background

2.1. SmolVLA

TODO: Sam Belliveau **(20%)**

General Overview: Explain the SmolVLA architecture.

- Discuss the motivation for small VLA models from the paper.
- Describe the overall pipeline: Visual Input -> Action Output.

SmolVLA is a novel Vision-Language-Action architecture designed to bridge the gap between high-level reasoning and low-level robot control. Traditional VLA models often rely on massive backbones that are impractical for edge deployment. SmolVLA addresses this by factoring the problem into two specialized components: a general-purpose Vision-Language Model (VLM) for reasoning and a lightweight "Action Expert" for trajectory generation.

The VLM processes the visual observations (from up to 3 cameras) and the user's natural language instruction to produce a high-level plan or "thought." This semantic representation is then fed into the Action Expert, which acts as a conditional diffusion policy to generate the sequence of joint actions required to execute the task.

2.1.1. Action Expert

TODO: Ezra Reiss **(20%)**

Action Expert Details:

- Explain the role of the Action Expert in the SmolVLA pipeline.
- Discuss how it interacts with the VLM/LLM components.

The Action Expert operates on a sequence of standard Transformer blocks but is optimized for the action generation domain. According to our configuration, the Action Expert operates with a hidden size of 720, 12 query heads, and 4 key/value heads (using grouped-query attention). The head dimension is set to 80, and the expert width is scaled to 0.75x relative to a standard VLM width.

The computational core consists of Cross-Attention layers, where query tokens (representing the robot's action plan) attend to the context provided by VLM embeddings, followed by MLP layers for feed-forward processing. The model uses a 10-step flow-matching solver to refine the action trajectory.

2.1.2. Large Language Model

TODO: Ezra Reiss **(20%)**

LLM Component:

- Describe the specific LLM used.
- Explain the prompt engineering or fine-tuning aspect.
- Discuss tokenization and embedding generation.

The VLM component of SmolVLA handles semantic scene understanding. It tokenizes input text and visual patches (64 tokens per frame) into a unified embedding space. The Language Model backbone within the VLM accounts for the largest parameter count in the entire architecture.

2.1.3. Vision Transformer Model

TODO: Sam Belliveau **(20%)**

ViT Details:

- Describe the Vision Transformer architecture (Patch Embedding, Attention Blocks).
- Mention specific parameters from `hardware_build/attention/config.py` (e.g., hidden size, number of heads).

The Vision Encoder is the primary focus of this acceleration effort. The visual front-end employs a Vision Transformer (ViT) to extract features from the camera inputs. These features are projected into the same embedding dimension as the text tokens, allowing the VLM to perform cross-modal reasoning.

2.2. Allo

Written By: Stanley Shen **(DONE)**

Allo is an accelerator design language that aims to simplify the process of designing accelerators on FPGAs. Developed by the Zhang Research Group at Cornell University, Allo decouples the functional aspects of a kernel from the hardware details and optimization code. In traditional HLS workflows, optimizing a kernel requires intrusive source edits to achieve performance improvements. Instead, Allo separates the functionality of a kernel from its schedule, allowing users to apply HLS optimizations without modifying the compute kernel itself.

A key feature of Allo utilized in this project is **Composable Transformations**, which allows us to apply optimizations like `.tile()`, `.pipeline()`, and `.partition()` as separate passes over the kernel. Additionally, its **Python-based DSL** enables kernels to be written in a Python subset, facilitating testing and integration with PyTorch-based implementations of SmolVLA. The framework's **Type System** supports reduced-precision data types (e.g., `int8`, `fixed<16, 8>`), which are crucial for maximizing the throughput of matrix multiplications on the U280's DSP slices.

2.3. Parallelization Schemes

2.3.1. Spatial Architectures

TODO: Ezra Reiss **(10%)**

Spatial Dataflow:

- Explain systolic arrays and dataflow architectures.

Spatial architectures, such as Systolic Arrays, are a natural fit for the dense matrix multiplications (GEMMs) found in Transformer attention and MLP layers. When working with memory-bound kernels, it is extremely important to utilize FIFO streaming between PEs to avoid off-chip HBM access. We utilized a spatial dataflow architecture.

2.3.2. Temporal Architectures

TODO: Ezra Reiss **(10%)**

Temporal Execution:

- Explain instruction-based execution.

Temporal architectures rely on SIMD (Single Instruction, Multiple Data) execution units where the same operation is broadcast to multiple data points. While flexible, they often require complex control logic to manage instruction scheduling. For our fixed-function Vision Encoder accelerator, we prioritized spatial dataflow to leverage the massive parallelism of the FPGA fabric.

3. Analytical Modeling Framework

TODO: Ezra Reiss **(0%)**

Framework Overview:

- Define the scope of analytical modeling (Roofline, resource bounds).
- Referenced `roofline_analysis/roofline_critique.md` for methodology.

Symbol	Definition
L	Sequence Length (Number of tokens)
D	Hidden Dimension
D_h	Head Dimension ($\frac{D}{\text{Heads}}$)
H_q	Number of Query Heads
H_{kv}	Number of Key/Value Heads
E	MLP Expansion Factor (typically 4)

Fig. 1: Summary of Analytical Model Dimensions and Symbols

3.1. Computational Demands

1. Vision Encoder (ViT) The Vision Encoder processes raw camera inputs using a standard 12-layer Vision Transformer architecture. This component handles 1024 patches per image, treating each 32×32 patch (derived from a 512×512 image) as a token. The model employs a hidden size (D) of 768, with 12 heads and an MLP expansion factor of 4x (resulting in an intermediate dimension of 3072).

2. VLM Backbone (Vision-Language Model) Fusing visual embeddings with text instructions, the VLM Backbone operates with a hidden size of 960. It processes a total of 113 tokens per camera—significantly fewer than the encoder—comprising 192 visual tokens, 48 text instruction tokens, and a single robot state token.

3. Action Expert The Action Expert generates control sequences via a conditional diffusion process (Flow Matching) over a prediction horizon of 50 action tokens. It executes 10 diffusion steps per inference using a 16-layer architecture that alternates between Self-Attention (even layers) and Cross-Attention (odd layers), where the latter attends to the VLM context. The model uses a hidden size of 720 (0.75x the VLM width) and employs Grouped Query Attention with 12 query heads and 4 key/value heads, each with a dimension of 80. The 50 Action Tokens interact with the 241 VLM Context Tokens through the odd-numbered Cross-Attention layers.

Compute Analysis Since our FPGA implementation utilizes `int8` quantization to maximize throughput on DSP slices, we quantify computational complexity in terms of Multiply-Accumulate operations (MACs) rather than FLOPs. A single MAC corresponds to one multiplication and one addition (effectively 2 ops if counting FLOPs).

The computational Demands are summarized by the expected MACs per token for a single Transformer layer. We distinguish between the Standard Multi-Head Attention (MHA) used in the Vision Encoder, and the Grouped Query Attention (GQA) used in the VLM Backbone and Action Expert.

Operation	MACs Formula	Notes
Q Projection	$L \cdot D^2$	Full Query Heads
K Projection	$L \cdot D^2 \cdot \left(\frac{H_{\text{kv}}}{H_q}\right)$	Reduced Heads
V Projection	$L \cdot D^2 \cdot \left(\frac{H_{\text{kv}}}{H_q}\right)$	Reduced Heads
Attn Scores	$L^2 \cdot D$	Broadcast K to matching Qs
Attn Update	$L^2 \cdot D$	Broadcast V to matching Qs
Output Proj	$L \cdot D^2$	Full Output
MLP FFN	$2 \cdot E \cdot L \cdot D^2$	Standard MLP
Total	$\approx LD^2\left(10 + 2\frac{H_{\text{kv}}}{H_q}\right) + 2L^2D$	Savings in K/V Proj

Fig. 2: Expected MACs for Grouped Query Attention Layer (GQA)

Operation	MACs Formula	Notes
Q Projection	$L \cdot D^2$	$D \times D$ weights
K Projection	$L \cdot D^2$	$D \times D$ weights
V Projection	$L \cdot D^2$	$D \times D$ weights
Attn Scores	$L^2 \cdot D$	QK^T (per head sum is D_h)
Attn Update	$L^2 \cdot D$	AV (per head sum is D_h)
Output Proj	$L \cdot D^2$	$D \times D$ weights
MLP FFN	$2 \cdot E \cdot L \cdot D^2$	Typically $8LD^2$ ($E = 4$)
Total	$\approx 12LD^2 + 2L^2D$	Dominated by linear layers

Fig. 3: Expected MACs for Standard Transformer Layer (MHA)

Methodology and Assumptions: Our MACs calculation assumes per-image processing for the Vision Encoder with an input sequence length of $L = 1024$ patches. For the VLM Backbone, we assume a single-camera mode with a sequence length of $L = 113$ (64 visual tokens + 48 text tokens + 1 state token). The Action Expert is modeled with a prediction horizon of $L = 50$ and 10 diffusion steps. Notably, we assume efficient KV reuse: the Cross-Attention Key/Value projections for the VLM context are computed only once per inference, while Query projections and Attention scores are computed at each diffusion step. The architecture uses Grouped Query Attention ($H_q = 12, H_{\text{kv}} = 4$) with a head dimension of $D_h = 60$.

Computational Demand Summary

Based on the parameters derived from the codebase and the specific configuration for this deployment (Single Camera, 113 VLM tokens), we calculate the total Multiply-Accumulate (MAC) operations per inference.

Crucially, for the **Action Expert**, we utilize a static optimization for the Cross-Attention layers: the Key and Value matrices for the VLM context are computed **once** per inference, as the context remains static across the 10 diffusion steps. Only the Query projections and the attention scores/updates are computed dynamically at each step.

Component	MACs (G)	OPs (G)	% of Total
Vision Encoder	106.30	212.60	60.9%
VLM Backbone	18.17	36.34	10.4%
Action Expert	50.05	100.10	28.7%
Total	**174.52**	**349.04**	**100%**

Fig. 4: Computational Demand Breakdown by Model Component

3.2. Resource Constraints

3.2.1. Compute Resource Constraints

Written By: Stanley Shen **(DONE)**

Fundamentally, most operations in SmolVLA can be reduced to matrix operations. These operations can in turn be broken down into multiply and addition steps, commonly called multiply and accumulate operations, or MACs. A naïve approach is to implement all of these operations directly in the FPGA fabric, synthesizing

them into LUTs and flip-flops. However, this can be highly inefficient because floating-point operations often require thousands of LUTs and flip-flops.

One way to reduce this overhead is to use lower-precision datatypes. The default floating-point format is FP32, which uses a whopping 4 bytes per value. By quantizing the model to FP16, bfloat16, FP8, or even FP4, we can significantly reduce memory usage while maintaining acceptable precision. Another approach is to convert the relatively complex FP32 values into integers. Integer ALUs require far fewer hardware resources than their floating-point counterparts, which makes them an appealing option for acceleration.

Another technique we use is mapping our MAC operations to DSP slices, which are hardened blocks on the FPGA designed to perform multiply and accumulate operations every cycle when pipelined. This saves valuable hardware resources and allows larger, more complex designs. On the AMD Alveo U280, there are 9,024 DSP slices, which means we can process at least 9,024 MAC operations per clock cycle with full utilization. However, we can use instantiate "soft" FPUs/ALUs on the LUT fabric, or we can use bit packing tricks to do up to 4 int4 MACs per clock cycle per DSP.

3.2.2. Memory Capacity Constraints

Written By: Ezra Reiss **(DONE)**

Memory Footprint Analysis

We analyze the storage requirements to determine where data must reside. The original model weights in `bfloat16` precision occupy approx. 897 MB. By quantizing to `int8`, we reduce the total model footprint to **448 MB**. This still exceeds the U280's on-chip capacity (40-50 MB), mandating off-chip HBM storage.

Metric	Size	Placement
Total Weights	359.08 MB	Off-Chip (HBM)
Peak Activations	1.57 MB	On-Chip (BRAM/URAM)
Action Context Cache	54.24 KB	On-Chip (Register/BRAM)

Fig. 5: Memory Footprint Requirements (Storage)

3.2.3. Memory Port Constraints

TODO: Ezra Reiss **(0%)**

Port/Bank Conflicts:

- Explain HLS partitioning constraints.
- Mention array partitioning directives used in Allo.

3.2.4. Memory Bandwidth Constraints

Theoretical Data Transfer Analysis

Due to the limited on-chip memory of the U280 (approx. 40-50MB URAM+BRAM) vs the large model size (approx. 180MB for weights), we assume a **layer-by-layer** execution model where weights must be streamed from HBM for each layer. For the Vision and VLM components, this means reading weights once per inference. However, for the **Action Expert**, the 10-step diffusion process requires re-streaming the dynamic weights 10 times, leading to a massive memory bandwidth demand.

Component	Transfer (MB)	Notes
Vision Encoder	103.81	Weights (1x) + I/O
VLM Backbone	160.76	Weights (1x) + I/O
Action Expert	937.29	Weights (10x) + I/O (10x)
Total	**1201.86**	Dominated by Action Loop

Fig. 6: Minimum Off-Chip Memory Transfer Per Inference (INT8)

Analysis: The Action Expert accounts for over 80% of the total off-chip data transfer. With a realistic HBM bandwidth of 300 GB/s, the memory transfer alone sets a hard lower bound on latency of approx. 4.6 ms ($1378 \frac{\text{MB}}{300}$ GB/s), not accounting for compute or latency hiding.

3.3. Performance Estimation

To evaluate the feasibility of our design on the Alveo U280, we first calculate the Operational Intensity (OI) for each major component. As summarized in Fig. 7, the Vision Encoder, VLM Backbone, and Action Expert all exhibit high operational intensities.

Component	OI (Ops/Byte)	Bound	Peak Perf
Vision Encoder	2048	Compute Bound	5.4 TOPS
VLM Backbone	226	Compute Bound	5.4 TOPS
Action Expert	103	Compute Bound	5.4 TOPS
U280 Ridge	**11.8**	—	—

Fig. 7: Operational Intensity and Hardware Limits

We visualize these characteristics against the hardware limits in the Roofline model shown in Fig. 2.

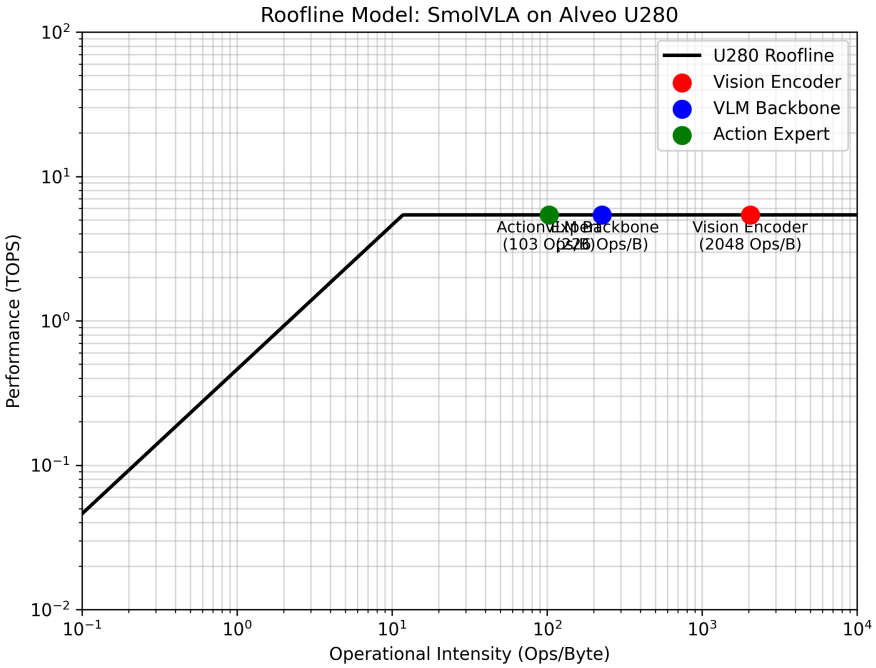

Fig. 2: **Roofline Analysis of SmolVLA on Alveo U280.** The plot visualizes the theoretical performance limits of the hardware. The kernel's Operational Intensity (Ops/Byte) places it in the compute-bound region, indicated by the horizontal roof, suggesting that performance is limited by available DSP resources rather than memory bandwidth.

Analysis: The Roofline analysis reveals that all three components of SmolVLA sit well to the right of the U280's ridge point (11.8 Ops/Byte). This indicates that the design is fundamentally **compute-bound**, limited by the DSP processing power rather than HBM bandwidth. The **Vision Encoder** is extremely compute-bound (OI 2048), suggesting that optimizing for DSP utilization (e.g., using systolic arrays) will yield direct performance gains. Similarly, the **Action Expert**, while having a lower OI (103) due to the requisite weight reloading for the diffusion process, remains in the compute-bound regime. However, it operates significantly closer to the memory wall; any inefficiency in the memory controller could easily shift this component into a bandwidth-bound regime.

3.3.1. Latency Estimation

TODO: Ezra Reiss **(0%)**

Latency Breakdown:

- Estimate latency per layer.
- Identify the bottleneck layer (Communication vs Computation).

3.3.2. Work Balancing

TODO: Sam Belliveau **(0%)**

Load Balancing:

- Discuss pipelining efficiency.
- Analyze if any stage is a significant bottleneck.

4. Overview of Workloads and Hardware

4.1. Latency of Different Stages

TODO: Ezra Reiss **(0%)**

Workload Characterization:

- Profile the runtime of the software baseline (if available).
- Break down latency by Attention vs MLP layers.
- Discuss the impact of sequence length (from `config.py`) on latency.

4.2. Quantization Schemes

TODO: Ezra Reiss **(0%)**

Quantization Strategy:

- Explain why int8/fixed-point is motivated (FPGA resource efficiency).
- Discuss specific quantization approach (Post-Training Quantization vs QAT).
- Reference any quantization scripts in `model-preparation`.

4.3. Memory Packing

TODO: Ezra Reiss **(0%)**

Data Layout:

- Explain how data is packed to maximize memory bandwidth (e.g., 512-bit packets).
- Discuss `pack`/`unpack` kernels if they exist in `hardware_build`.

4.4. Memory Bandwidth

TODO: Ezra Reiss **(0%)**

Bandwidth Requirements:

- Reiterate bandwidth constraints specifically for the workload.
- Discuss effectiveness of caching or specific memory hierarchy decisions on the U280.

5. Implementations

5.1. Allo Kernels

TODO: Ezra Reiss **(0%)**

General Kernel Structure:

- Explain how kernels are defined in Allo.
- Discuss common optimization patterns applied (tiling from `schedule` functions in `matrix_multiplies.py`).
- Discuss the systolic array implementation if applicable.

5.2. Accelerating Attention Layers

TODO: Ezra Reiss (0%)

Attention Implementation:

- Detail `hardware_build/attention/self_attention`.
- Explain the Q, K, V matrix multiplication chain.
- Discuss the specific bottleneck in Softmax and how it's handled on FPGA.

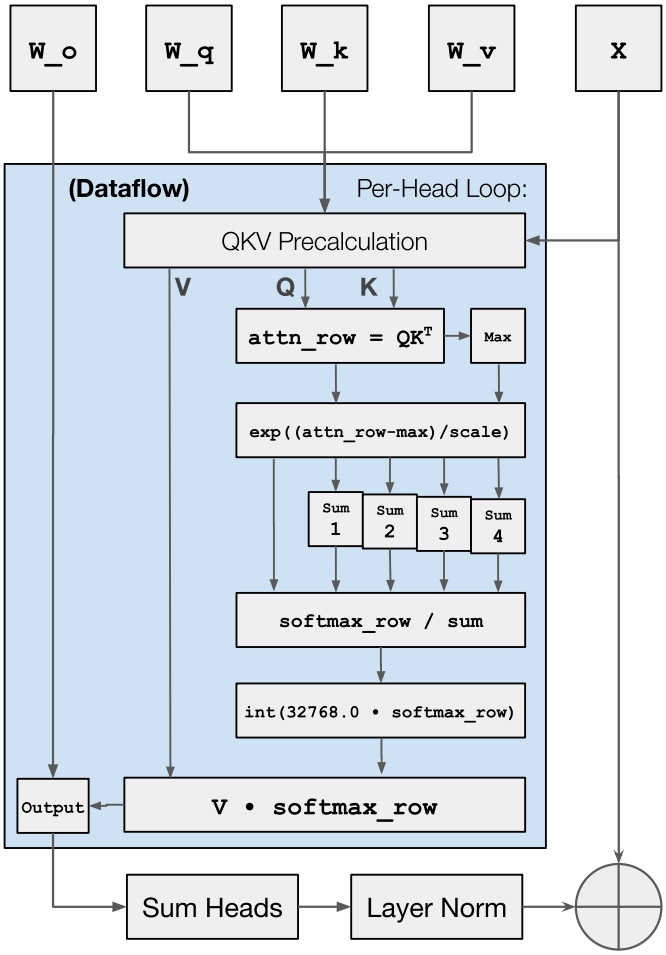

Fig. 3: **Dataflow diagram of the Allo-implemented Attention kernel for SmolVLA.** The diagram details the Per-Head Loop, including QKV precalculation, scaled dot-product attention with soft max, and the specific quantization step before the final multiplication with the value (V) vector. The results from each head are then summed and passed through Layer Norm.

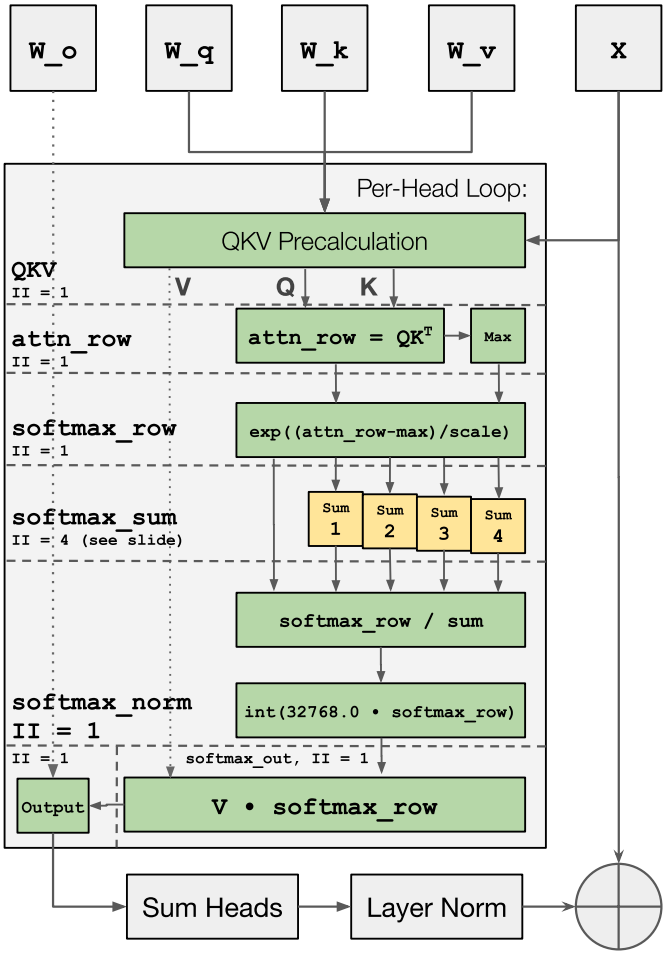

Fig. 4: **Pipelined dataflow architecture of the Attention kernel with scheduling annotations.** The diagram demarcates distinct pipeline stages within the per-head loop, explicitly listing the Initiation Interval (II) for each. Most stages, including QKV precalculation and final value multiplication, achieve a high-throughput II of 1, while the soft max sum reduction operates at an II of 4.

5.3. Accelerating MLP Layers

TODO: Stanley Shen (99%)
MLP Implementation: FILL IN THE ERF FORMULA

The MLP pipeline comprises a fully connected (FFN) layer followed by a Gaussian Error Linear Unit (GELU) non-linear activation function. We selected GELU over other common activation functions primarily for its smoothness and differentiability, which improve stability and information preservation in smaller models. The output is then passed to a second fully connected layer before entering the layer normalization stage.

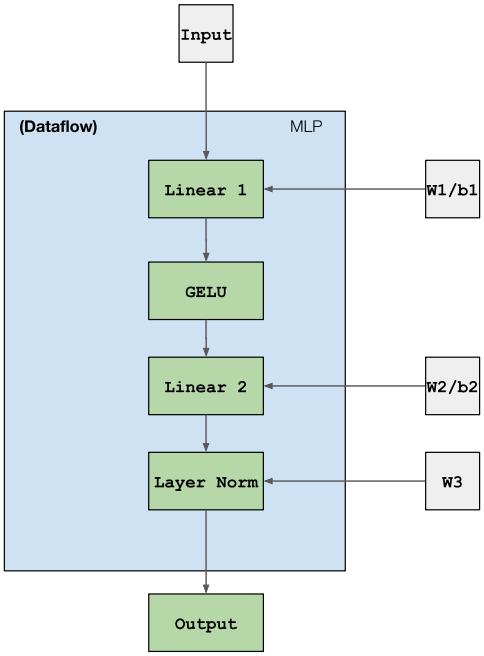

Fig. 5: **Diagram of the MLP in SmolVLA's vision encoder** The diagram details how data flows into the MLP from a high level perspective.

We compute the linear layer by multiplying input tensors with weight tensors and adding bias vectors. The primary challenge lies in the size of these tensors. Of the 9.6 billion MACs in the MLP, 99.6% are attributed to these two large matrix multiplications. In contrast, the 8 billion MACs in the Self-Attention mechanism are distributed across 72 smaller matrix multiplications (12 heads $\times$ 6 multiplications per head).

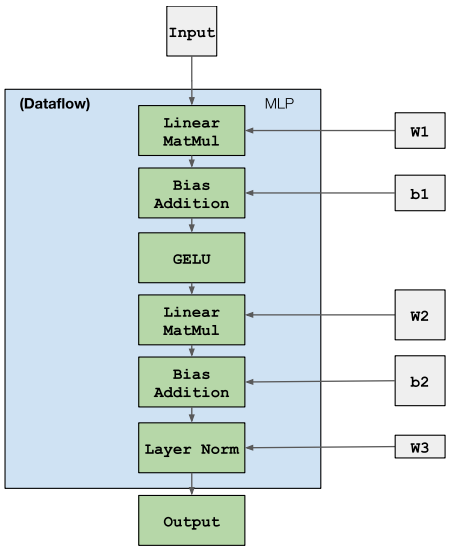

Fig. 6: **Diagram of the MLP operations in SmolVLA's vision encoder** The diagram details how data is broken down into matrix operations.

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

Another optimization target is the GELU calculation. The standard GELU formula involves the Error Function (erf), which requires computing an integral—an operation ill-suited for FPGA hardware. As a result, we approximate GELU using a hyperbolic tangent (tanh) formulation:

$$\text{GELU}_{\approx(x)} = \frac{1}{2} * x * \left(1 + \tanh\left(\sqrt{\frac{2}{\pi}} * (x + 0.044715 * x^3)\right)\right)$$

. We express the tanh function using a polynomial approximation based on Cody and Waite's rational form. This approach requires only 4 floating-point multiplications (fmul), 3 additions (fadd), and 1 division (fdiv) in single precision. Combined with the non-tanh operations (2 fadd, 6 fmul, 1 fdiv), the entire GELU calculation requires just 16 operations.

Simpler approximations exist, such as the sigmoid approximation

$$\text{GELU}(x) \approx x * \sigma(1.702 * x)$$

. However, we did not employ them as the MLP runtime is dominated by matrix multiplication. We did, however, experiment with replacing GELU with ReLU to isolate and test the matrix multiplications without activation function bottlenecks.

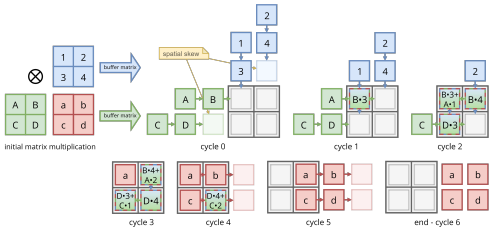

Fig. 7: **Systolic Array Architecture.** A 2D array of Processing Elements (PEs) that passes data rhythmically (systolically) across rows and columns. This architecture maximizes data reuse for matrix multiplications, minimizing memory bandwidth requirements. (Wikipedia)

For the matrix multiplications, the standard way to execute these are by unrolling and pipelining the triple nested loop. We will also experiment with a systolic array based implementation. With this approach, data is injected into the edge processing elements, and then only moves between processing elements. This reduces the amount of data movement needed between the memory/buffers, helping increasing utilization of all of the DSPs on the FPGA.

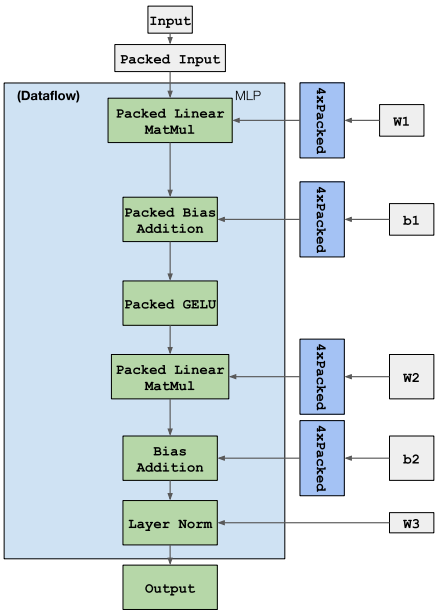

Fig. 8: **Data Packing Strategy for MLP Weights.** To optimize memory bandwidth, 8-bit integer weights are packed into 32-bit words, allowing four weights to be transferred per clock cycle. This format aligns with the DSP slice capabilities and reduces the number of required memory ports.

We also implemented weight and tensor packing. Since our weights and activations are 8-bit integers (`int8`), we can pack up to four values into a single 32-bit word. This optimization is crucial for BRAM data transfers, which typically operate on 32-bit words. Without packing, non-consecutive memory accesses could require up to four separate read cycles. Packing guarantees that we can move four weights per cycle. This creates a path for future optimizations using AXI for off-chip HBM transfers, where reducing the number of data beats per transaction alleviates memory bandwidth constraints.

6. Evaluation

6.1. Evaluation of Attention Layers

TODO: Ezra Reiss **(0%)**

Attention Metrics:

- Report exact latency (cycles/ms) for the Self-Attention kernel.
- Report resource usage (DSP, BRAM, LOOT, FF) from Vivado reports.
- Compare against the analytical model predictions.

6.1.1. Ablation

TODO: Ezra Reiss **(0%)**

Attention Ablation:

- Compare baseline (unoptimized) vs tiled vs systolic array versions.
- Explain which optimization yielded the biggest gain.

Kernel	Speed (ms)	BRAM %	DSP %
Baseline	TODO	TODO	TODO
Tiled	TODO	TODO	TODO
Systolic	TODO	TODO	TODO

Fig. 8: **Ablation Study of Attention Kernels.** Performance progression from the unoptimized baseline to the fully optimized implementation. Key metrics include inference latency (ms) and resource consumption (BRAM, DSP) on the U280 FPGA.

6.2. Evaluation of MLP Layers

Written By: Stanley Shen **(DONE)**

To evaluate the MLP, we estimated latency by measuring the cycle count for a single query execution. Resource utilization was derived from Vitis synthesis reports. Specifically, we tracked Look-Up Table (LUT), Flip-Flop (FF), DSP slice, and Block RAM (BRAM) consumption to quantify the FPGA resource usage.

6.2.1. Ablation

TODO: Stanley Shen **(90%)**

MLP Ablation:

- Show progression of optimizations for MLP.

Kernel FC1/FC2	Activation	Latency (ms)	BRAM %	LUT %	DSP %
Systolic 1x1	GELU	8055	8485 (210%)	19653 (1%)	68 (0%)
Systolic 1x1	RELU	8055	8480 (210.32%)	11035 (0.85%)	4 (0.04%)
Systolic 12x30, 40x30	RELU	30.29	4,209 (104.39%)	8,386,130 (643.27%)	631 (6.99%)
Tiled 1x1	RELU	8055	8,480 (210.32%)	11,035 (0.85%)	4 (0.04%)
Tiled 12x30, 40x30	RELU	25.05	3,894 (96.58%)	43,115 (3.31%)	59 (0.65%)

Fig. 9: **Ablation Study of MLP Kernels.** Comparison of latency and resource utilization across different optimization strategies. The optimized systolic array implementation with RELU approximation achieves a significant latency reduction compared to the baseline, fitting within the target resource budget.

7. Discussion

TODO: Ezra Reiss **(0%)**

Synthesis of Results:

- Discuss specific bottlenecks encountered (e.g., Self-Attention Softmax).
- Comment on the efficacy of HLS high-level synthesis vs RTL for this workload.

7.1. Performance of Attention

TODO: Ezra Reiss (0%)

Attention Insights:

- Analyze why specific optimizations (tiling, unrolling) worked or didn't.
- Discuss memory bandwidth saturation.

7.2. Performance of MLP

TODO: Stanley Shen (0%)

MLP Insights:

- Discuss the specific challenges of the MLP layers (large weight matrices).
- Resource trade-offs found during implementation.

Several architectural optimizations were explored for the MLP. Our baseline design did not utilize tiling or any optimization techniques. This version had an extremely high latency, as there was no pipelining, resulting in serial execution of all operations. Consequently, we synthesized varying dimensions of the systolic array. As the dimensions of the systolic array increase, latency decreases due to higher parallelism in the matrix multiplication. We varied the aspect ratios of our systolic array dimensions to maximize utilization. However, Allo's current systolic array implementation proved inefficient.

To address this, we implemented tiling, exploiting temporal reuse and dataflow control. The MLP computation is partitioned into tiles, allowing the same hardware to be reused across multiple tiles over time. This dramatically reduced resource utilization and allowed for synthesis of a feasible design with significantly reduced latency. The synthesis results for estimated latency and resource utilization are shown in Figure 9.

The main contributors to the latency for the MLP are the two fully connected layers, FC1 and FC2, as they account for the majority of the MAC operations. It can be noted that latency will scale approximately linearly with batch size regardless of these optimizations, and that batch size does not have a direct impact on resource utilization.

7.3. Fused Kernel Performance

TODO: Ezra Reiss (0%)

Future Work/Fusion:

- Feasibility of fusing Attention and MLP layers.
- Potential performance gains from kernel fusion (reducing off-chip memory access).

8. Related Work

TODO: Ezra Reiss (0%)

Literature Review:

- Cite recent FPGA accelerators for Transformers (e.g., FTRANS, etc.).
- Discuss other VLA/VLM acceleration efforts.
- Contrast with GPU implementations of similar small models.
- Mention recent works using Allo or similar MLIR-based flows.

9. Conclusion

TODO: Sam Belliveau (0%)

Final Remarks:

- Summarize key findings: "Allo enables rapid prototyping of VLA accelerators…"
- Reiterate the main performance numbers (speedup vs baseline).
- Conclude on the viability of FPGAs for SmolVLA edge deployment.