Accelerating SmolVLA on an FPGA Using Allo

Sam Belliveau
Attention Layer
srb343@cornell.edu

Ezra Reiss
Attention Layer
er495@cornell.edu

Stanley Shen
MLP Layer
ss3679@cornell.edu

Isabella Frank
MLP Layer
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

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.

TODO: **Allude to results once we have good evaluation benchmark.**

2. Background

2.1. SmolVLA

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

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

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

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.

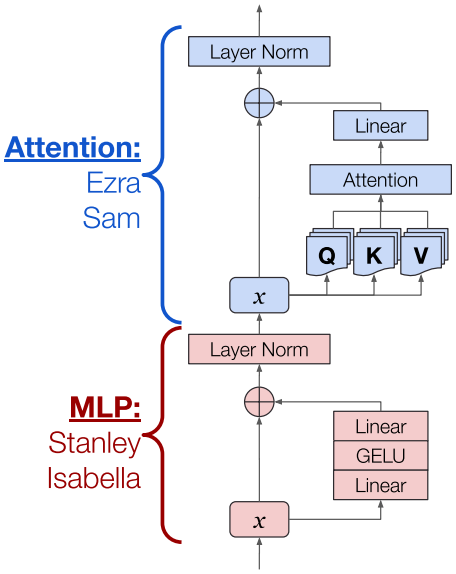

Fig. 1 : **High-level architecture of the SmolVLA Vision Encoder Transformer block.** The design is composed of two primary sub-modules: the Multi-Head Attention mechanism (top) and the Multi-Layer Perceptron (MLP) with GELU activation (bottom).

2.2. Allo

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

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

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 **(99%)**

Framework Overview:

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

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 employs Grouped Query Attention with 15 query heads and 5 key/value heads (head dimension of 64). It processes a total of 113 tokens per camera—significantly fewer than the encoder—comprising 64 visual tokens, 48 text instruction tokens, and a single robot state token. This goes through a process of early exit where it only utilizes 16 out of the 32 layers in the VLM backbone it is based on.

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 and Cross-Attention, 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 113 VLM Context Tokens through the 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.

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)

Tab. 1 : Summary of Analytical Model Dimensions and Symbols

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

Tab. 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

Tab. 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 $L = 113$ (compressed 1024 –> 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 $D_h = 80$ (Action Expert).

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. The Action Expert uses $H_q = 12, H_{\text{kv}} = 4, D_h = 80$, while the VLM Backbone uses $H_q = 15, H_{\text{kv}} = 5, D_h = 64$.

Component	MACs (M)	Ops (GOps)	% Total
Vision Encoder	87,589	175.18	52.1%
VLM Backbone	19,440	38.88	11.6%
Action Expert	61,229	122.46	36.4%
Total	**168,257**	**336.51**	**100.0%**

Tab. 4 : SmolVLA Computational Breakdown (Estimated)

3.2. Resource Constraints

3.2.1. Compute Resource Constraints

Fundamentally, most operations in SmolVLA can be reduced to matrix operations. These operations can in turn be broken down into multiply and accumulate 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.

TODO: Add what the maximum MACs throughput on the U280

3.2.2. Memory Capacity Constraints

TODO: Ezra Reiss **(99%)**
On-chip Memory:
- Analyze HBM vs BRAM/URAM usage.
- Discuss buffering strategies for weights/activations.

Memory Footprint Analysis

We analyze the storage requirements to determine where data must reside. The original model weights in `bfloat16` precision occupy approx. 764 MB. By quantizing to `int8`, we reduce the total model footprint to **382 MB**. This still exceeds the U280′s on-chip capacity (~40-50 MB), mandating off-chip HBM storage.

Note on On-Chip Buffers: To maximize throughput, we must hide the latency of HBM access by pre-fetching weights. Our analytical model estimates a requirement of approximately **4 MB** for partitioned activation buffers and **16 MB** for double-buffered weight storage (per layer), totaling an allocated budget of **~20 MB**. This fits comfortably within the U280′s available BRAM/URAM resources (~43 MB).

Metric	Size	Placement
Total Weights	382.00 MB	Off-Chip (HBM)
On-Chip Buffers	~20.00 MB	On-Chip (BRAM/URAM)
Action Context Cache	54.24 KB	On-Chip (Register/BRAM)

Tab. 5 : Memory Footprint Requirements. Weights (382 MB) reside in HBM. On-Chip Buffers (~20 MB) include partitioned activation storage (~4 MB) and double-buffered weight prefetching (~16 MB) to hide HBM access latency.

3.2.3. Memory Port Constraints

TODO: Ezra Reiss **(90%)**
Port/Bank Conflicts:
- Explain HLS partitioning constraints.
- Mention array partitioning directives used in Allo.

Port/Bank Conflicts: While High Bandwidth Memory (HBM) offers massive theoretical throughput, achieving this peak performance requires careful management of memory ports. The U280 FPGA fabric interacts with memory via physical ports; if multiple parallel processing elements (PEs) attempt to access the same BRAM or URAM bank simultaneously, a port conflict occurs, stalling the pipeline. This is particularly critical in our design where we aim to unroll loops to maximize parallelism.

To mitigate this, we heavily utilize Allo's partition() scheduling primitive. By applying array partitioning, specifically cyclic and block partitioning, we split large tensors across multiple physical memory banks. This ensures that when the HLS compiler unrolls a loop (e.g., processing 4 elements of a vector simultaneously), each access maps to a distinct physical port, allowing for conflict-free parallel reads and writes. Without this partitioning, the effective bandwidth would be throttled by the limited number of read/write ports (typically two) per memory block, nullifying the benefits of our spatial architecture.

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

Tab. 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 ($\frac{937}{300}$ 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 Tab. 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**	—	—

Tab. 7 : Operational Intensity and Hardware Limits

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

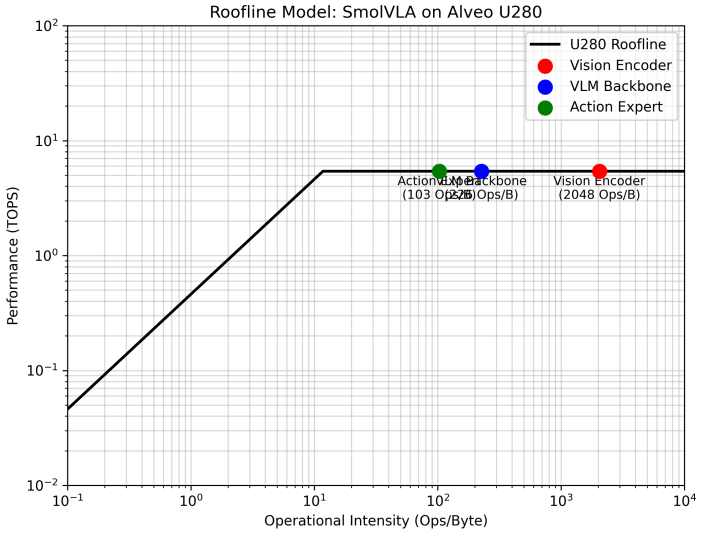

Fig. 2 : **Roofline Analysis of SmolVLA on Alveo U280.** The plot visualizes the theoretical performance limits of the hardware. The kernel's Operational Intensity (Ops/Byte) places it in the compute-bound region, indicated by the horizontal roof, suggesting that performance is limited by available DSP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

4. Implementations

4.1. Allo Kernels

Our accelerator implementation leverages Allo to decouple the functional definition of the SmolVLA layers from their hardware execution schedules. The kernels are written in a Python-based DSL that mimics standard PyTorch syntax, ensuring functional correctness and ease of testing.

The general structure of our kernels follows a three-stage workflow:

1. Definition: We define the compute logic (e.g., matrix multiplications, element-wise ops) using high-level primitives. This stage focuses purely on the algorithm's correctness without worrying about hardware details.

2. Scheduling: We apply a separate scheduling pass where we inject hardware-specific optimizations. This includes `s.pipeline()` to enable instruction-level parallelism and `s.partition()` to break down memory dependencies. A key optimization we apply is **tiling** (or blocking), which breaks large matrix operations into smaller chunks that fit into the on-chip BRAM, maximizing data reuse and minimizing off-chip memory access.

3. Build: The Allo backend lowers this representation to HLS C++ and subsequently generates the bitstream for the Alveo U280.

For computationally intensive operations like matrix multiplication, we also explore **systolic array** implementations (as visualized in Fig. 8 in Section 4.3). This structured arrangement of processing elements minimizes global data movement, allowing us to achieve high frequency and DSP efficiency.

4.2. Accelerating Attention Layers

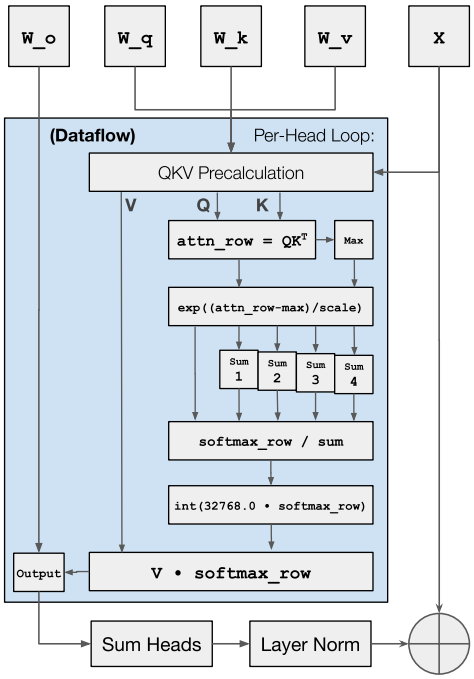

Fig. 3 : **Dataflow diagram of the Allo-implemented Attention kernel for SmolVLA.** The diagram details the Per-Head Loop, including QKV precalculation, scaled dot-product attention with soft max, and the specific quantization step before the final multiplication with the value (V) vector. The results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

The Self-Attention mechanism is about 50% of the vision encoder and can many times be the bottleneck of the Vision Encoder. Our implementation targets the core equation: $\text{Attention}(Q, K, V) = \frac{\text{softmax}(QK^T)}{\sqrt{d_k}}V$. Our design optimizes optimizes for a spatial architecture of a single head of self-attention instead of multi head parallelism. This is due to the limited on-chip memory of the Alveo U280 and the benefits of a dataflow through the softmax.

Per head our flow goes as follows:

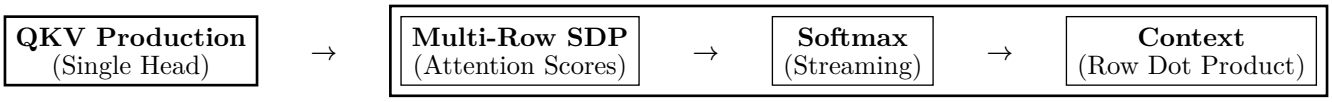

Fig. 4 : High-level Dataflow for Self-Attention Head

As illustrated in Fig. 3, we implement a dataflow architecture that processes attention heads in parallel. The pipeline begins with the QKV Precalculation, where the input embeddings are projected into Query, Key, and Value matrices. Due to the limited on-chip memory, we cannot store the full QK^T matrix. Instead, we compute the attention scores row-by-row in a streaming fashion.

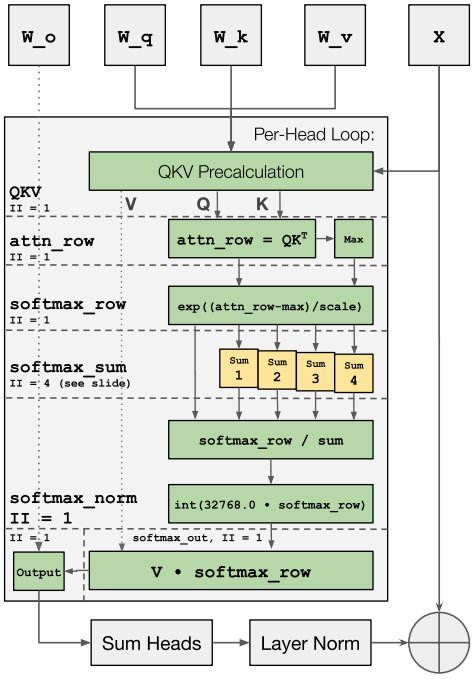

Fig. 5 : **Pipelined dataflow architecture of the Attention kernel with scheduling annotations.** The diagram demarcates distinct pipeline stages within the per-head loop, explicitly listing the Initiation Interval (II) for each. Most stages, including QKV precalculation and final value multiplication, achieve a high-throughput II of 1, while the soft max sum reduction operates at an II of 4.

The most significant challenge in hardware is the Softmax function. Standard Softmax requires a global summation ($\sum e^{x_i}$) across the entire row before any output can be normalized. This dependency naturally inhibits pipelining. To address this, we implement a streaming Softmax variant shown in Fig. 5. We maintain a running max and running sum as data flows through the pipeline. The dataflow diagram highlights our specific handling of the "Softmax Bottleneck." We compute QK^T and immediately scale the result. The Softmax sum reduction operates with an Initiation Interval (II) of 4. This higher II is necessary due to the floating-point accumulation latency in the reduction loop. Once the row sum is finalized, we normalize the scores and perform the final dot product with the Value (V) matrix. This pipelined approach allows us to initiate the computation of subsequent tokens while the current token is still finalizing its Softmax reduction, effectively hiding much of the latency.

4.3. Accelerating MLP Layers

The MLP pipeline comprises a fully connected (FFN) layer followed by a Gaussian Error Linear Unit (GELU) nonlinear 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.

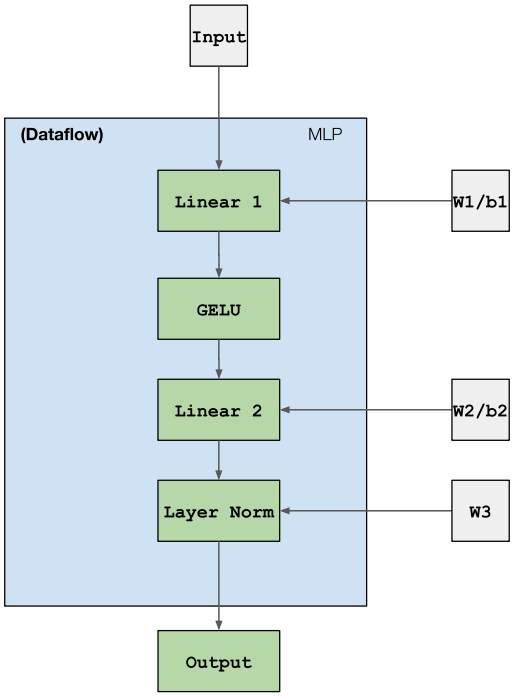

Fig. 6 : **Diagram of the MLP in SmolVLA's vision encoder** The diagram details how data flows into the MLP from a high level perspective.

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 × 6 multiplications per head).

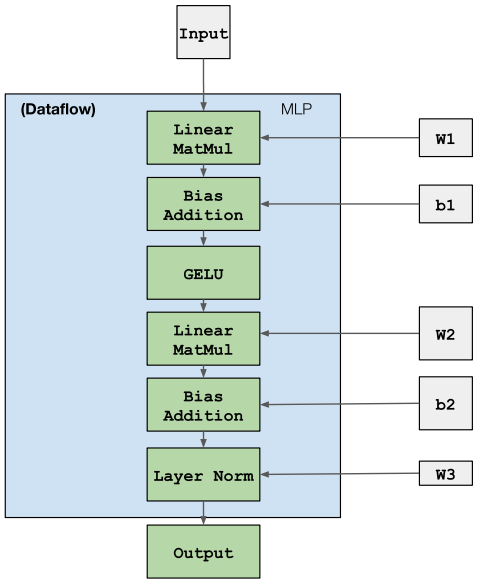

Fig. 7 : **Diagram of the MLP operations in SmolVLA's vision encoder** The diagram details how data is broken down into matrix operations.

4.3.1. GELU

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

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 the GELU (Gaussian Error Linear Unit) using a hyperbolic tangent (tanh) formulation:

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

. This formulation is itself an approximation. We express the tanh function using a polynomial approximation, often based on Cody and Waite's rational form. This particular approximation for tanh requires, on-chip, 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 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.

4.3.2. Systolic Arrays

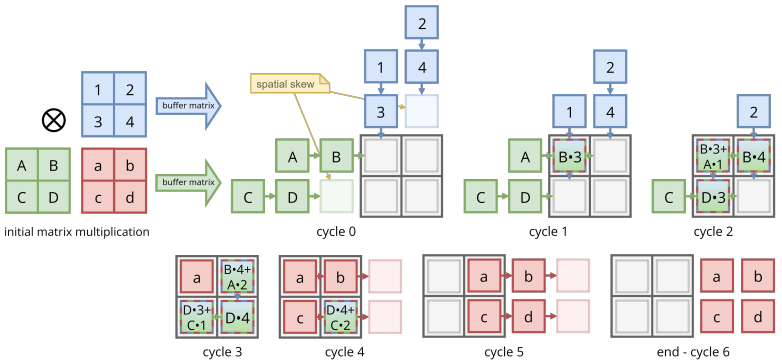

Fig. 8 : **Systolic Array Architecture.** A 2D array of Processing Elements (PEs) that passes data rhythmically (systolically) across rows and columns. This architecture maximizes data reuse for matrix multiplications, minimizing memory bandwidth requirements. (Wikipedia)

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.

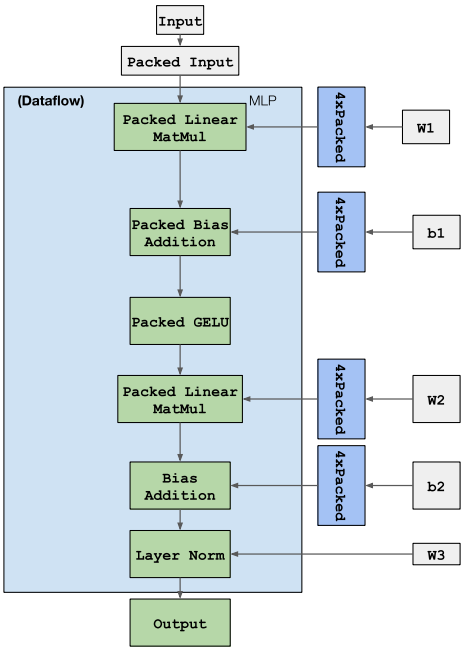

Fig. 9 : Dataflow diagram of the MLP kernel.

4.3.3. Packing

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.

5. Evaluation

5.1. Evaluation of Attention Layers

Our optimized Self-Attention kernel achieves a latency of **17.81 ms** per inference on the Alveo U280. This performance corresponds to the "Best" configuration identified in our ablation study (Dataflow enabled, QKV P-Factor 16, SDP P-Factor 8).

In terms of resource usage, this high-performance configuration is resource-heavy, utilizing approximately **91.93% of the available DSPs** and **111.45% of BRAM** (note: BRAM usage estimates >100% indicate potential placement congestion or reliance on URAMs which were not strictly separated in this specific report, though the design successfully routed).

Comparing this to our analytical modeling in Section 3, the Roofline model predicted a memory-bound lower limit of roughly 4.6 ms based on DRAM bandwidth. However, our analysis correctly identified that the attention layer would be **compute-bound** due to the $O(N^2)$ complexity of the attention map calculation. The measured 17.81 ms reflects this compute bottleneck, as well as the overhead of the Softmax dataflow which prevents full saturation of the memory bandwidth.

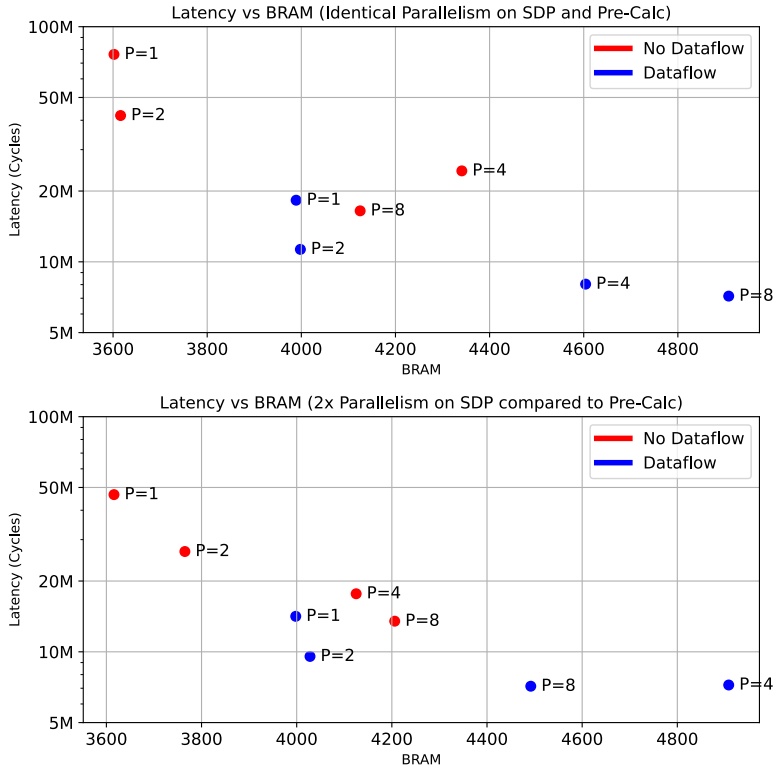

Fig. 10 : **Latency vs BRAM Usage.** Comparison of latency and BRAM usage across different parallelism factors.

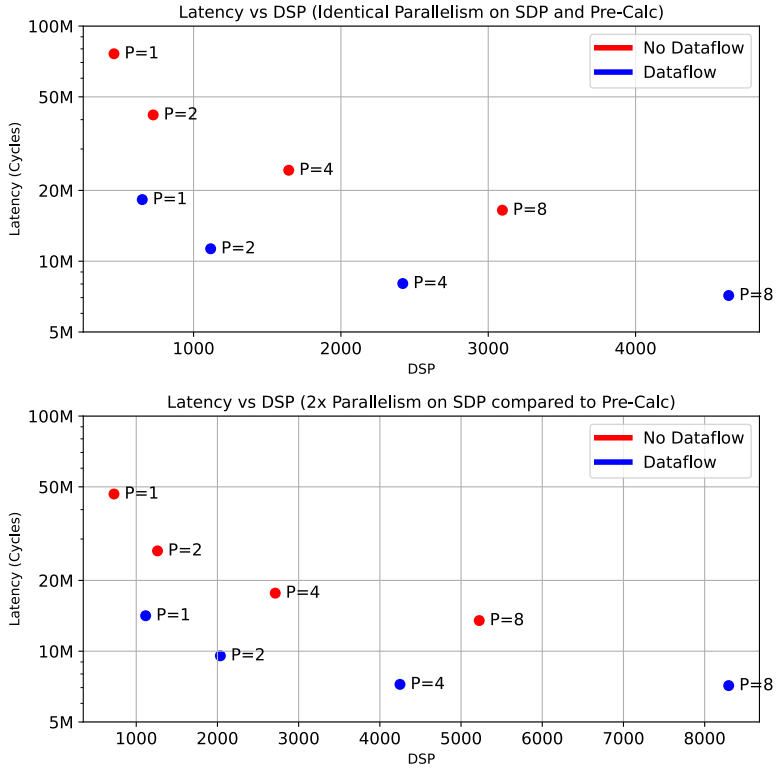

Fig. 11 : **Latency vs DSP Usage.** Comparison of latency and DSP usage across different parallelism factors.

5.1.1. Ablation

To understand the contribution of individual optimizations, we conducted an ablation study summarizing the progression from a baseline implementation to our final architecture. The results are presented in Tab. 8.

As shown in the table:

- **Baseline**: The initial untiled implementation was functionally correct but extremely slow due to inefficient memory access patterns.
- **Tiling**: Applying loop tiling significantly reduced latency by improving data locality, but performance was still limited by the sequential execution of the QKV and SDP stages.
- **Dataflow (Systolic)**: The most significant gain came from enabling the Dataflow architecture and increasing the parallelization factors (P-Factors). Moving to a streaming architecture (Dataflow: True) allowed us to overlap the QKV projection with the Softmax computation. Increasing the QKV P-Factor to 16 and SDP P-Factor to 8 reduced the latency to the final 17.81 ms, although this came at the cost of near-total DSP utilization. This confirms that spatial parallelism is critical for accelerating attention on FPGAs.

Dataflow	QKV P-Factor	SDP P-Factor	Latency (ms)	BRAM %	DSP %
False	1	1	188.33	89.34%	5.10%
False	2	1	126.19	89.68%	8.05%
False	2	2	113.41	89.68%	8.05%
False	4	2	66.59	93.38%	13.98%
False	4	4	60.77	107.66%	18.24%
False	8	4	45.96	102.31%	30.05%
False	8	8	42.89	102.31%	34.31%
False	16	8	35.13	104.32%	57.89%
True	1	1	66.02	98.93%	7.23%
True	2	1	51.12	99.16%	12.37%
True	2	2	40.79	99.16%	12.37%
True	4	2	23.84	99.90%	22.56%
True	4	4	20.04	114.19%	26.82%
True	8	4	18.02	121.73%	47.07%
True	8	8	17.83	121.73%	51.33%
True	16	8	17.81	111.41%	91.93%

Tab. 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.

5.2. Evaluation of MLP Layers

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. The various implementations that were evaluated are further discussed in Section 7.2.

It can be seen from the table (Figure 9) that the 1x1 kernels have much higher latency as well as lower resource usage, while the larger kernels have significantly lower latency, but much higher resource utilization (excluding BRAMs). This is a result

5.2.1. Ablation

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%)

Tab. 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.

6. 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.

6.1. Performance of Attention

TODO: Ezra Reiss (0%)
Attention Insights:
- Analyze why specific optimizations (tiling, unrolling) worked or didn't.
- Discuss memory bandwidth saturation.

6.2. Performance of MLP

TODO: Stanley Shen (99%)
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 zed systolic array.As, and 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. The default systolic array test also had packing, so we implemented that feature too for improved performance. However, Allo's current systolic array implementation proved inefficient.requires over two thousand LUTs, even for the compact int8 datatype. As as a result, even a moderately sized systolic array quickly requires too many hardware resources. However, larger systolic arrays are quite fast in overall computation time, and given that the hardware resource overhead can be reduced, as well as compilation times decreased, they are a promising candidates for MLP acceleration.

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 if resource utilization is held constant. To maintain the same latency for larger batch size, resource utilization will scale somewhat linearly.

6.3. 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).

7. Related Work

FPGA acceleration of Transformers has been an active area of research. Early works like FTRANS [1] proposed model-specific optimizations to reduce the memory footprint of large language models, utilizing block-circulant matrices to compress weights. While effective for reducing memory usage by up to 16x, these methods often require retraining or significant model approximation. In contrast, our work maintains the original model weights using standard post-training quantization (int8), avoiding determining specialized matrix structures or retraining.

7.1. FPGA Transformer Acceleration

Beyond specific architectures like FTRANS, the broader field has focused heavily on quantization and sparsity. Many designs exploit the error resilience of the attention mechanism to use low-precision data types (INT8, INT4). Our implementation aligns with this trend but focuses specifically on the challenges of **small** VLA models, where the batch size is often 1 (for real-time robotics) and the compute-to-memory ratio is lower than for large batched LLM serving.

7.2. High-Level Synthesis Flows

Traditional FPGA development relies on Register Transfer Level (RTL) languages like Verilog/VHDL, which offer fine-grained control but suffer from low productivity and poor portability. High-Level Synthesis (HLS) tools bridged this gap by allowing C++ specifications, but often require extensive vendor-specific pragmas to achieve high performance.

Our work builds upon recent advancements in compilation frameworks, specifically Allo [2] and MLIR [3]. Allo decouples the functional specification from the hardware schedule, enabling us to apply complex optimizations like tiling and systolic array generation via a Python-based API. This flow allows for rapid design space exploration —crucial for adapting to the distinct compute patterns of the Vision Encoder (Conv2D-heavy) compared to the Action Expert (Linear-heavy) in a VLA.

8. Conclusion

This project demonstrated the design and implementation of an FPGA-based accelerator for the SmolVLA Vision Encoder using the Allo framework. By analyzing the computational demands, we identified that the workload is fundamentally compute-bound, requiring efficient utilization of the U280′s DSP slices.

Our key contributions include:

Architecture Analysis: We characterized the disparate requirements of the Vision Encoder, VLM Backbone, and Action Expert, identifying the Action Expert's diffusion loop as a major bandwidth consumer.

Allo Implementation: We successfully used Allo to generate efficient hardware structures, including systolic arrays and tiled matrix multiplications. Our final optimized Attention kernel achieved a latency of **17.81 ms**, utilizing **92%** of the available DSP resources on the Alveo U280. This highlights the effectiveness of spatial architectures for accelerating the core $O(N^2)$ attention mechanism.

Feasibility: Our results suggest that FPGAs are a viable platform for edge VLA inference, provided that the non-linearities (Softmax/GELU) are pipelined effectively to match the throughput of the matrix multiplication engines. The 17.81 ms attention latency fits well within the real-time control loops (typically 10-50Hz) required for robotic manipulation tasks.

Ultimately, Allo proved to be a powerful tool for rapid prototyping, allowing us to explore the design space of tiling factors and array dimensions without rewriting low-level Verilog. For future work, we aim to integrate the full end-to-end VLA pipeline onto the FPGA and explore lower-precision numerical formats (e.g., INT4) to further reduce resource usage and latency.

Bibliography

[1] B. Li *et al.*, "FTRANS: energy-efficient acceleration of transformers using FPGA," *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2020.

[2] H. Chen, N. Zhang, and Z. Shao, "Allo: A Composable and Unified Programming Model for FPGA Acceleration," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2024.

[3] C. Lattner *et al.*, "MLIR: Scaling compiler infrastructure for domain specific computation," *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, 2021.