Embedded System for Aerial Image Segmentation with Transformers

João Barreiros C. Rodrigues

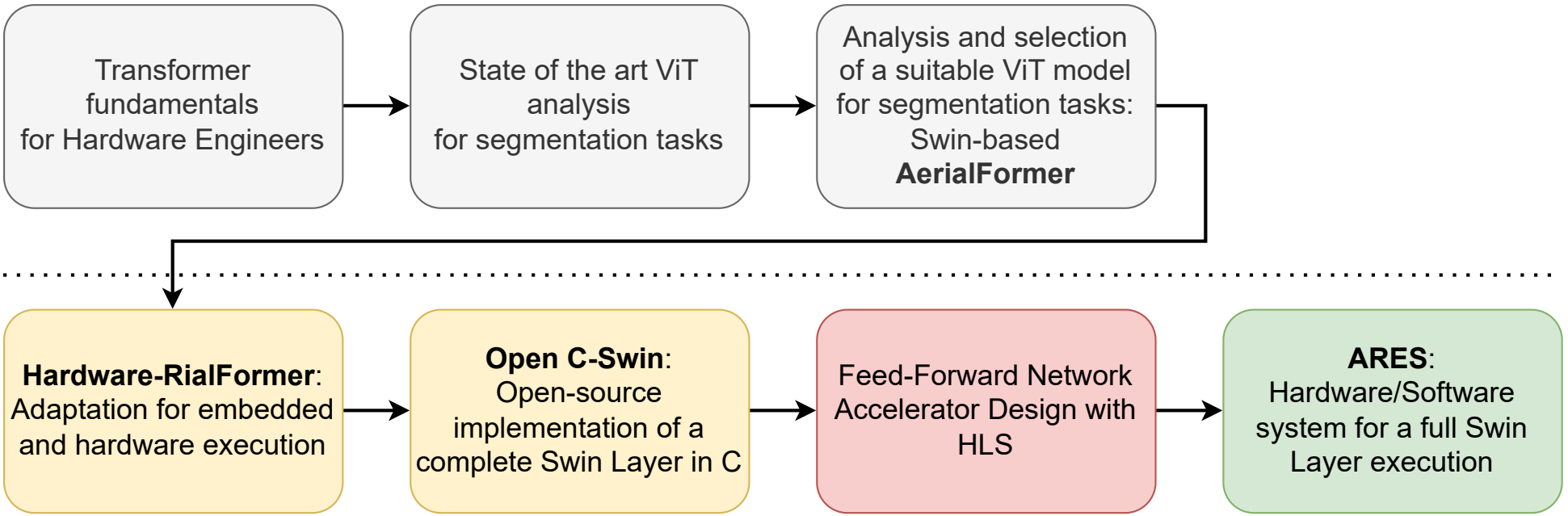
Técnico-ULisboa

Objectives

This work aims to:

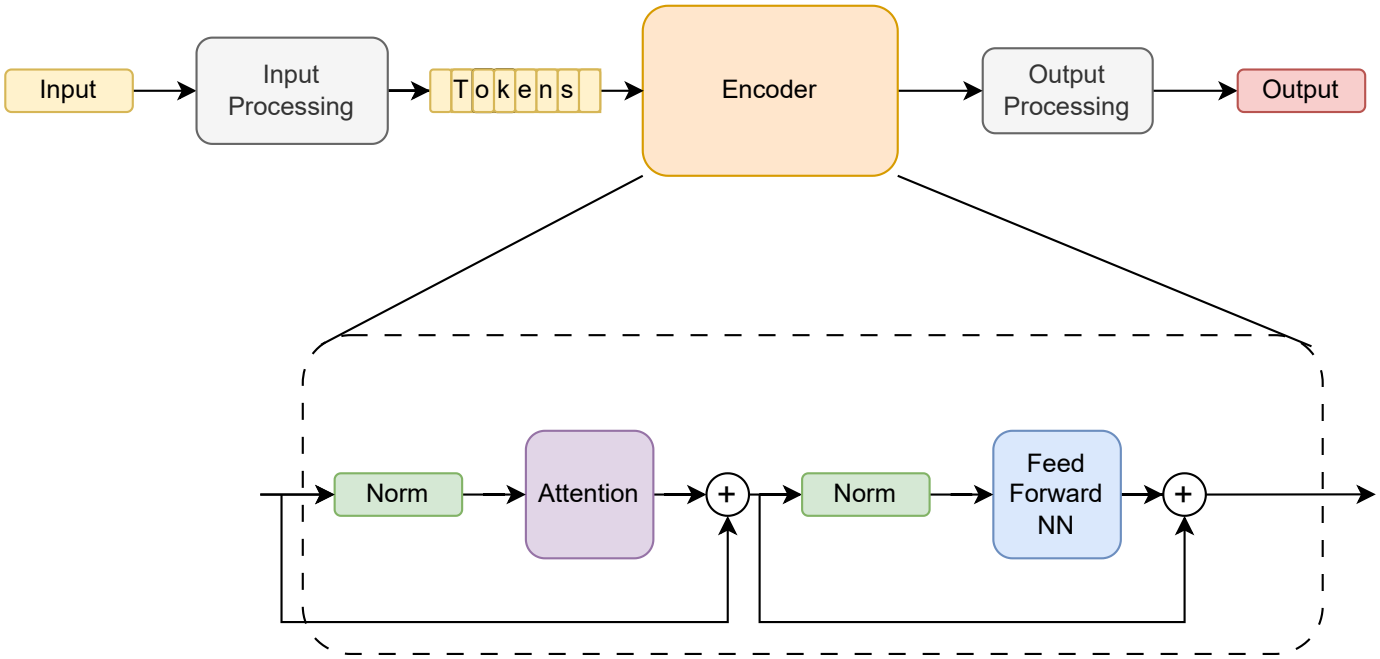
- Examine **Vision Transformers** (ViT) **parts** and **bottlenecks**
- Choose a **ViT model** for aerial image segmentation
- Adapt the **ViT** for embedded execution
- Implement **key ViT parts** in open-source embedded C
- Design **hardware IP** for primary bottleneck
- Build a **Hw/Sw system** proof of concept in SoC-FPGA

Outline

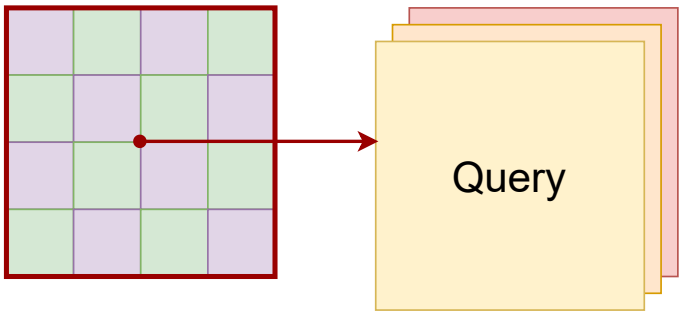

Transformer fundamentals for Hardware Engineers
State of the art ViT analysis for segmentation tasks
Analysis and selection of a suitable ViT model for segmentation tasks: Swin-based **AerialFormer**
Hardware-RialFormer: Adaptation for embedded and hardware execution
Open C-Swin: Open-source implementation of a complete Swin Layer in C
Feed-Forward Network Accelerator Design with HLS
ARES: Hardware/Software system for a full Swin Layer execution

Generic Transformer Architecture

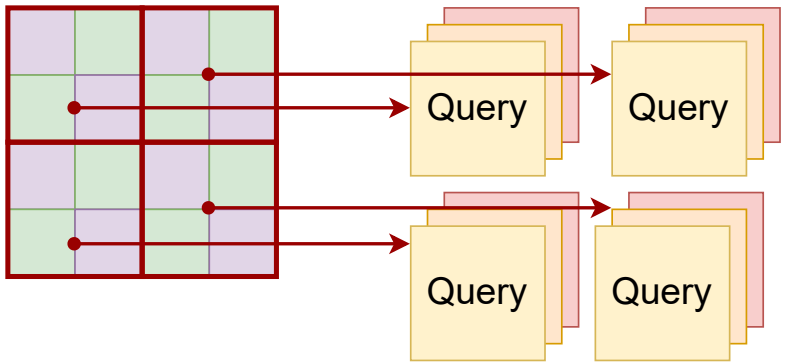
- **Input processing** (tokenization)
- **Encoder**
  - Attention
  - Feed-Forward Network
- **Output processing**



Attention - QKV generation



Self-Attention
(SA)



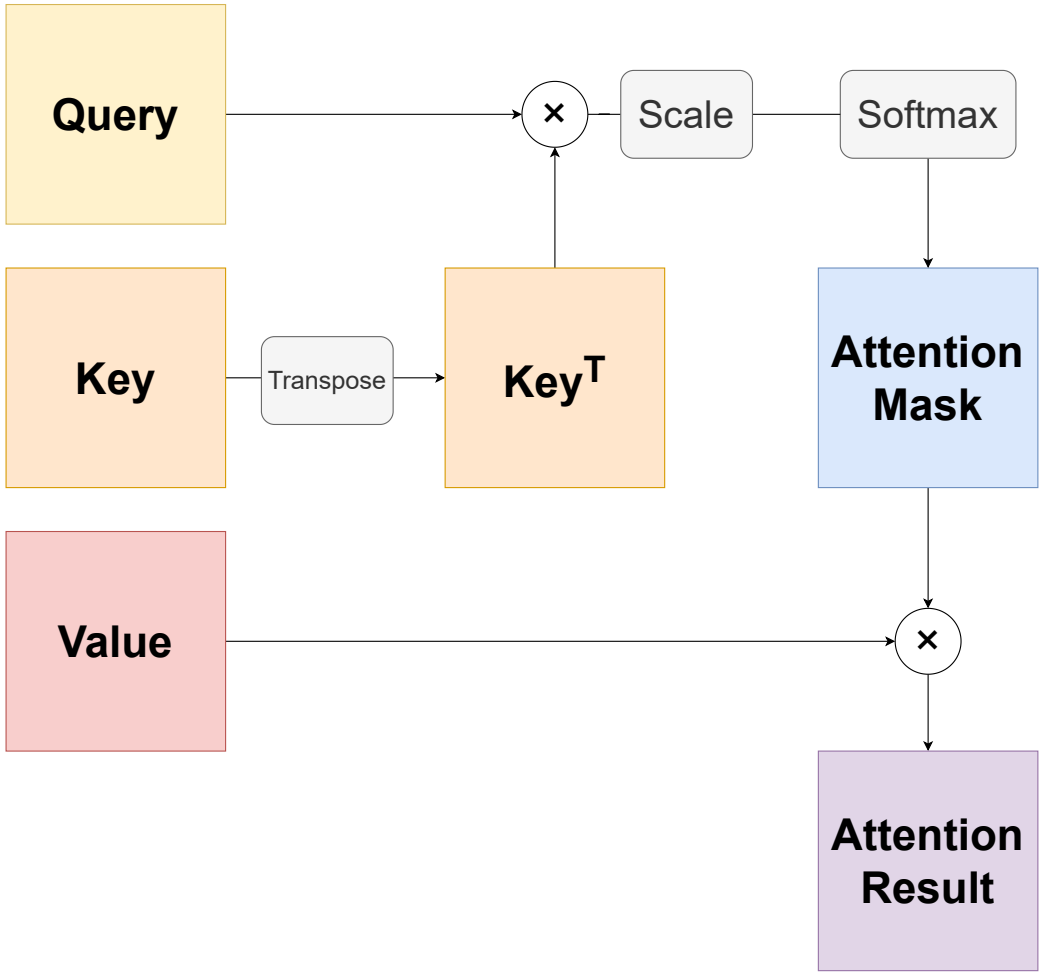
Window Self-Attention
(WSA)

In attention 3 matrices are extracted from the input:

- **Query** → Represents the currently attended token
- **Key** → Represents all other tokens
- **Value** → Represents the data of the tokens

Attention - Attention Mechanism

The QKV matrices are then combined to generate the attention result.



Feed-Forward Network

Feed-Forward Network $\rightarrow$ cascade of two linear transforms, with a activation function in between:

$$\mathrm{FFNN}(\mathrm{Result}) = \phi(\mathrm{Result} \times W_1 + b_1)\, W_2 + b_2$$

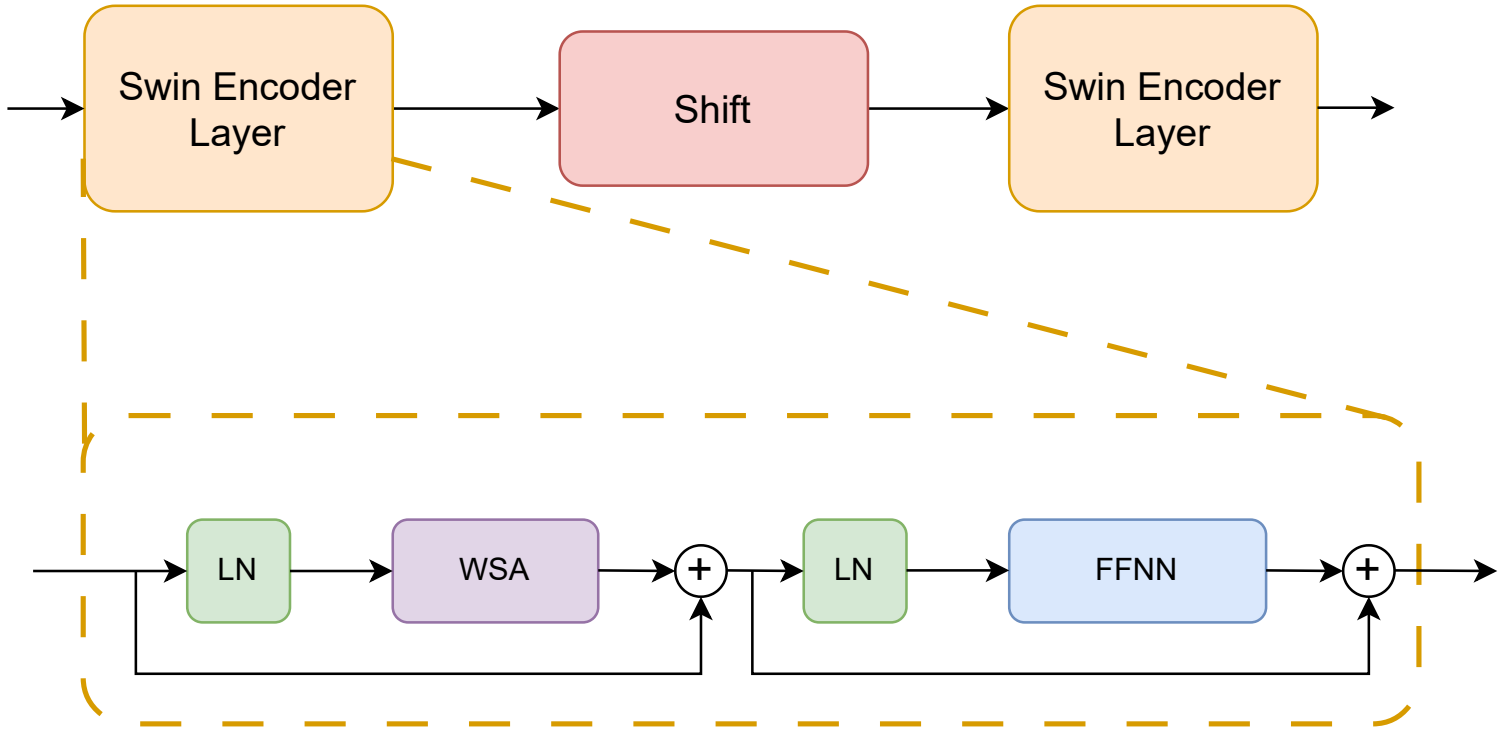
where $\phi(x)$ is tipically:

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

The FFNN ensures an **efficient feature extraction**, avoiding recurrent activation of the same features.

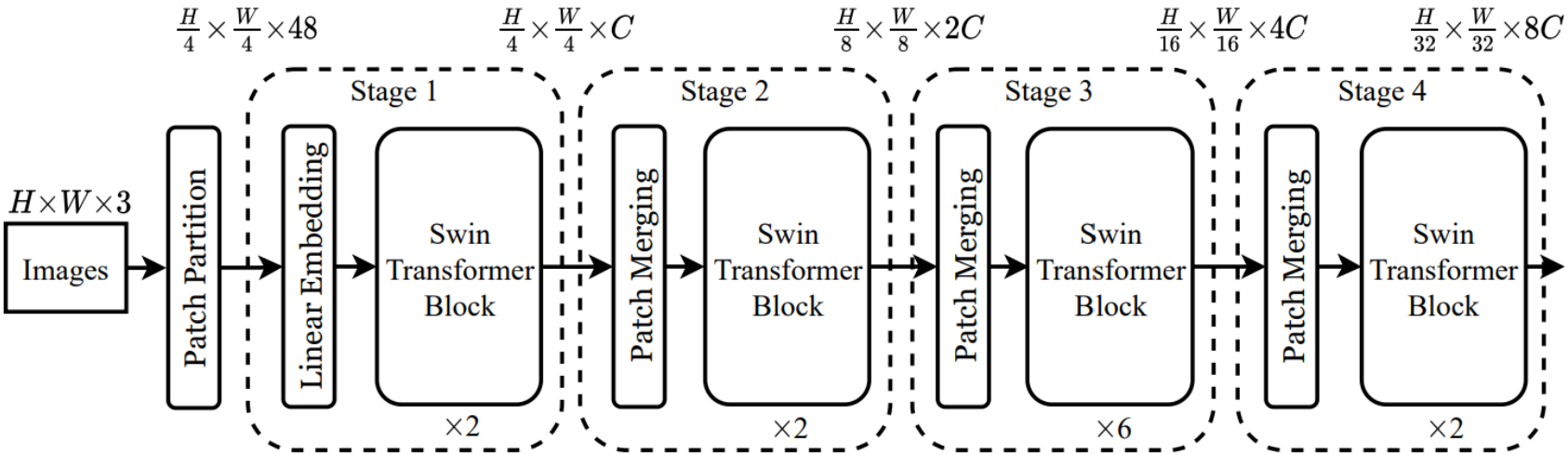
Swin Block

To provide global context using WSA, the window location is shifted between each two consecutive Swin Layers:

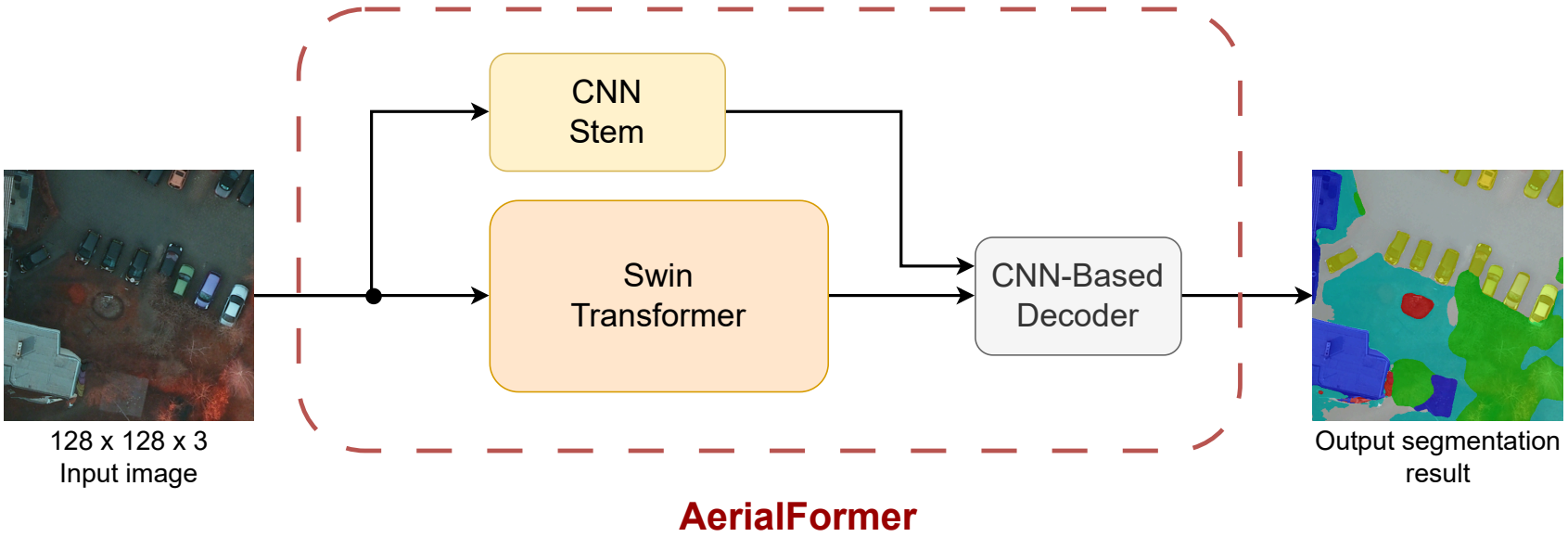


Swin Transformer

Microsoft's Swin was the first Transformer to implement WSA as its attention mechanism.



AerialFormer



The AerialFormer model is designed for aerial image segmentation, using the **Swin Transformer** as the encoder.

Hardware-RialFormer

The AerialFormer has 3 main components which are bottlenecks in hardware and in embedded systems computation:

- **GELU** $\rightarrow$ Complex computation of `erf()` or `tanh()`
- **LayerNorm** $\rightarrow$ Recomputation of normalization parameters for each channel, at runtime
- **Softmax**$\rightarrow$ Computation of `exp()`

Original	**Tentative replacement**
$\mathrm{GELU}(x) = x \cdot \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$	$\mathrm{Hardswish4}(x) = x \cdot \frac{\mathrm{ReLU4}(x+2)}{4}$
$\mathrm{LayerNorm}(x) = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}\gamma + \beta$	$\mathrm{BatchNorm}(x) = \frac{x-\mu_{\mathrm{fixed}}}{\sqrt{\sigma^2_{\mathrm{fixed}}+\epsilon}}\gamma + \beta$
$\mathrm{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$	$\mathrm{Squaremax}(x_i) = \frac{\mathrm{ReLU}(x_i)^2}{\sum_j \mathrm{ReLU}(x_j)^2+\epsilon}$

Hardware-RialFormer - Results

Various experiments were made to find suitable replacements:

	Experiment	OA (%) (Overall Accuracy)	mIoU (%) (mean Intersection Over Union)
	Software Baseline	88.82	74.76
(1)	Hardswish4 (Hs4) + SimpleNorm	88.84	74.70
(2)	Hardswish (Hs)	88.75	74.52
(3)	Hs + SimpleNorm	88.62	74.26
(4)	FFNNorm	88.26	73.56
(5)	Hs + FFNNorm	87.94	73.15
(6)	SquareMax (SqMax)	87.34	71.90
(7)	Hs + SqMax	87.03	71.42
(8)	Hs + FFNNorm + SqMax	84.54	67.17

experiment (1) was adopted.
New Architecture $\rightarrow$ **Hardware-RialFormer**
Full Swin Layer $\rightarrow$ Convert to C

Hardware-RialFormer - Results

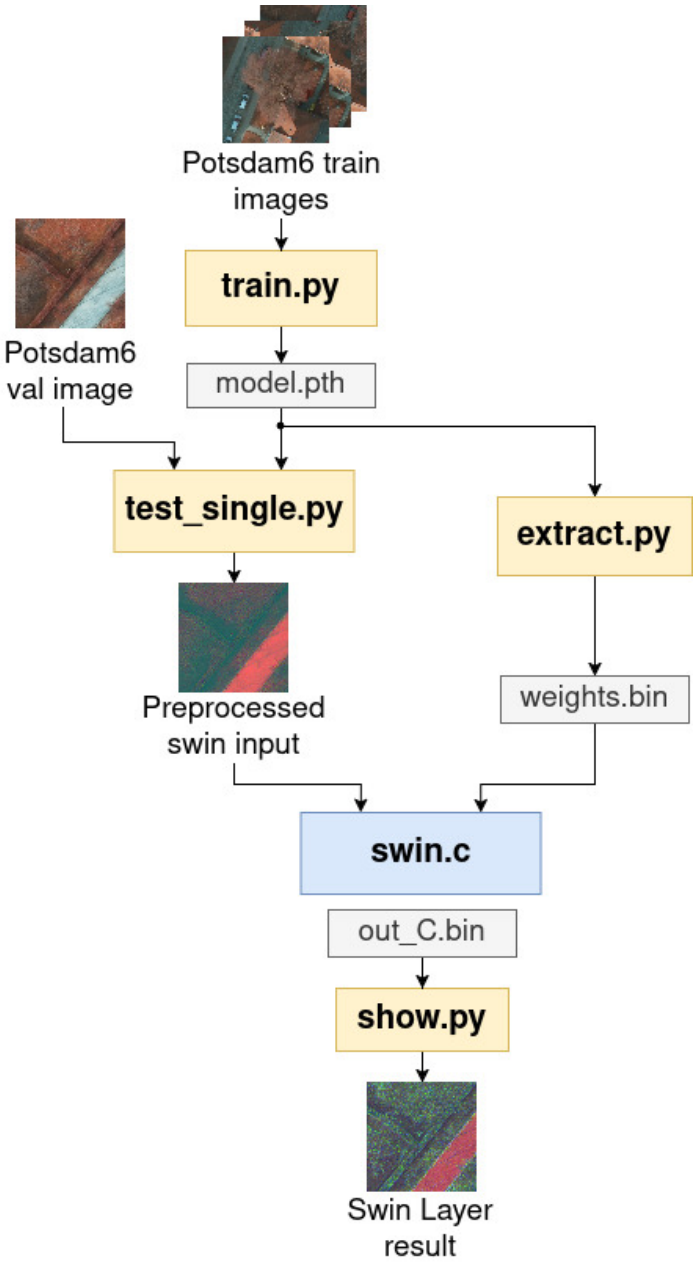
Various experiments were made to find suitable replacements:

	Experiment	OA (%) (Overall Accuracy)	mIoU (%) (mean Intersection Over Union)
	Software Baseline	88.82	74.76
(1)	Hardswish4 (Hs4) + SimpleNorm	88.84	74.70
(2)	Hardswish (Hs)	88.75	74.52
(3)	Hs + SimpleNorm	88.62	74.26
(4)	FFNNorm	88.26	73.56
(5)	Hs + FFNNorm	87.94	73.15
(6)	SquareMax (SqMax)	87.34	71.90
(7)	Hs + SqMax	87.03	71.42
(8)	Hs + FFNNorm + SqMax	84.54	67.17

experiment (1) was adopted.
New Architecture $\rightarrow$ **Hardware-RialFormer**
Full Swin Layer $\rightarrow$ Convert to C

Workflow

- **train.py** → Trains the model
- **extract.py** → Extracts weights and bias
- **test_single.py** → Extracts pre-processed input and validation binaries
- **swin.c** → Runs inference for a full Swin Layer
- **show.py** → Uses PCA to vizualize `.bin` files

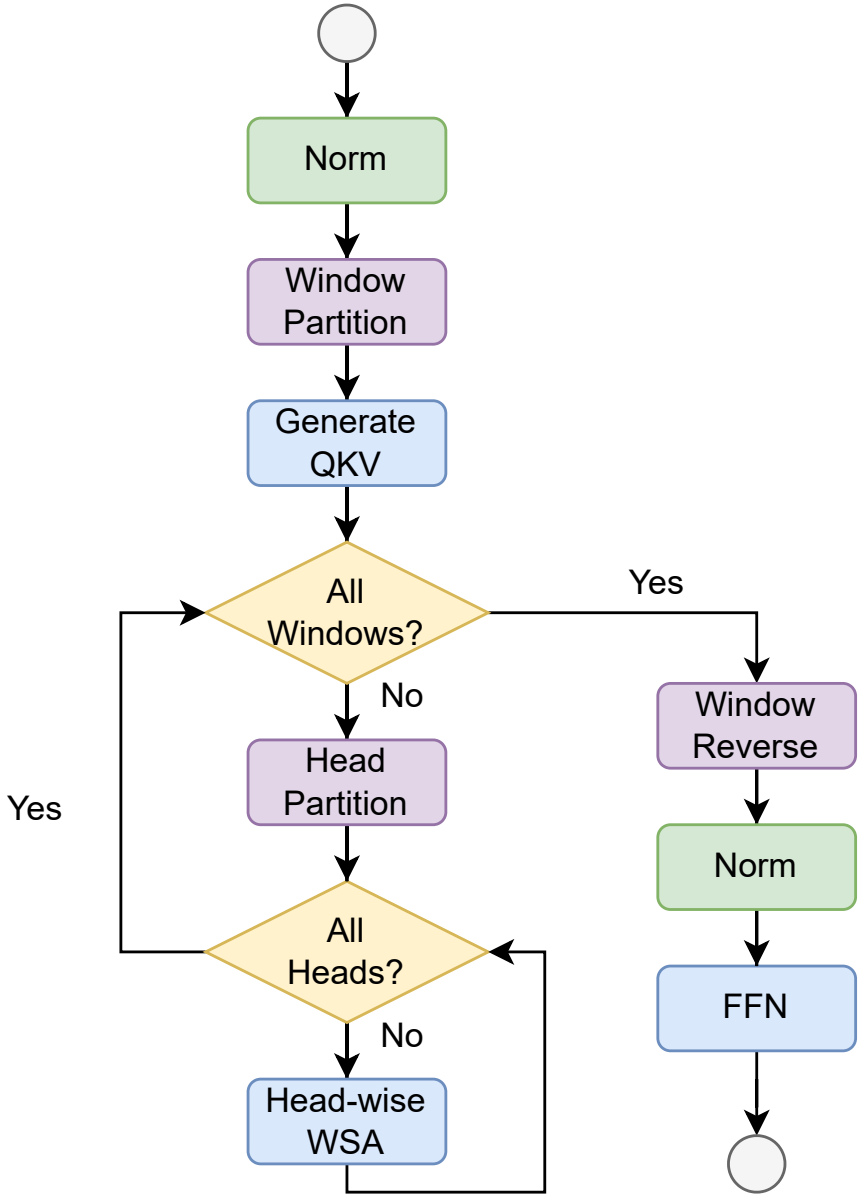


swin.c Flowchart

The **embedded-oriented** Swin Layer implements:

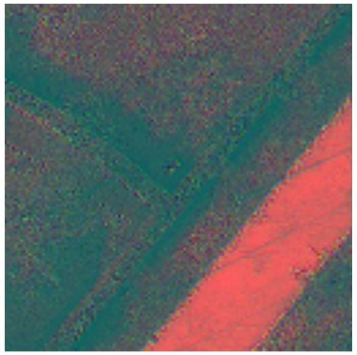
- MatMul and MatAdd
- 1D BatchNorm
- Hardswish4
- Softmax
- **Window Partition**
- **Window Reshape**
- **WSA** routine
- **FFN** routine

open-source C model, for the first time in literature.

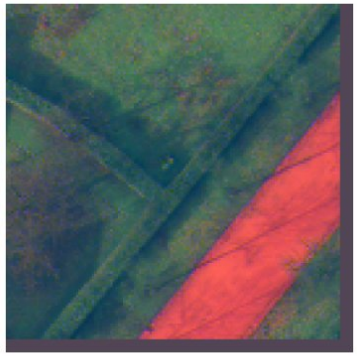


swin.c Validation

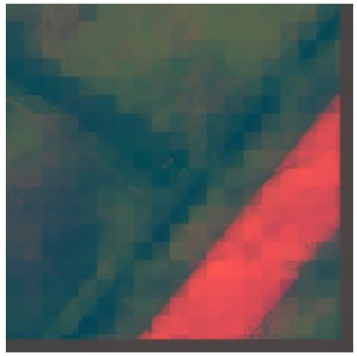
Debugging and **Validation** $\rightarrow$ Direct binary comparison with Python intermediate and final results



Swin Layer input

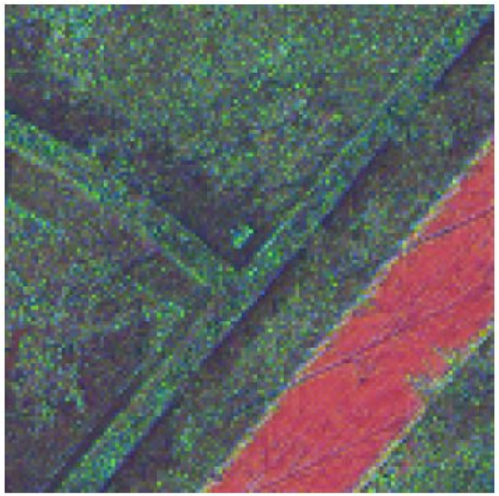


post-normalization and post-padding

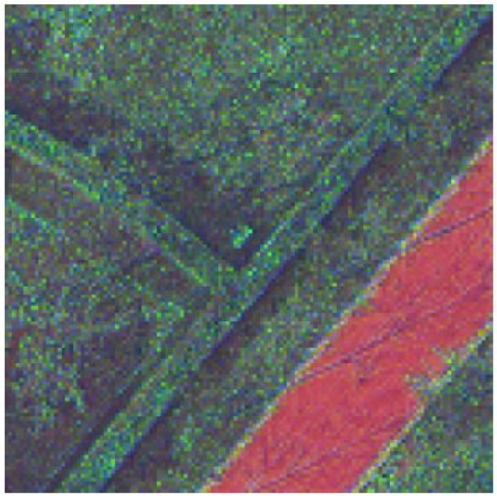


attention result

PCA Visualization



Swin Layer output - C implementation



Swin Layer output - expected Python result

swin.c Profiling

The C application was executed on embedded ARM processor
(Ultrascale+ SoC-FPGA on ZCU104 board)

	Non-Optimized (-O0)		Optimized (-O2)	
Sub Component	**Time (s)**	**Time (%)**	**Time (s)**	**Time (%)**
QKV Gen.	30.7	23.9	5.1	23.9
Swin Attention	11.3	8.8	2.0	9.4
FFN	75.6	58.9	12.4	58.2
Others	10.8	8.4	1.8	8.5
Total	128.4	100	21.3	100

FFN routine $\rightarrow$ takes around **58%** of the **execution time** $\rightarrow$ primary target for **hardware acceleration**

swin.c Profiling

The C application was executed on embedded ARM processor (Ultrascale+ SoC-FPGA on ZCU104 board)

	Non-Optimized (-O0)		Optimized (-O2)	
Sub Component	**Time (s)**	**Time (%)**	**Time (s)**	**Time (%)**
QKV Gen.	30.7	23.9	5.1	23.9
Swin Attention	11.3	8.8	2.0	9.4
FFN	75.6	58.9	12.4	58.2
Others	10.8	8.4	1.8	8.5
Total	128.4	100	21.3	100

FFN routine $\rightarrow$ takes around **58%** of the **execution time** $\rightarrow$ primary target for **hardware acceleration**

Swin FFN Quantization - Results

Quantization (w, a)	OA (%)	mIoU (%)
Float32	88.8	74.7
(16,16)	84.2	66.5
(8,8)	82.2	63.7
(8,4)	65.3	40.9
(4,8)	64.9	39.5
(4,4)	63.9	38.6

Selected **8 bits** for **both activations and weights**.

Swin FFN Quantization - Results

Quantization (w, a)	OA (%)	mIoU (%)
Float32	88.8	74.7
(16,16)	84.2	66.5
(8,8)	82.2	63.7
(8,4)	65.3	40.9
(4,8)	64.9	39.5
(4,4)	63.9	38.6

Selected **8-bits** for **both activations and weights**.

Quantized swin.c Profiling

	Non-Optimized (-O0)		Optimized (-O2)	
Sub Component	**Time (s)**	**Time (%)**	**Time (s)**	**Time (%)**
QKV Gen.	30.7	25.8	5.1	18.3
Swin Attention	11.3	9.5	2.0	7.2
FFN	66.1	55.5	18.8	67.6
Others	11.1	9.3	1.9	6.8
Total	119.2	100	27.8	100

The quantized **FFN** optimized routine takes around **68%** of the execution time $\rightarrow$ conversions between `int` and `float` benefit little from opt.

Quantized swin.c Profiling

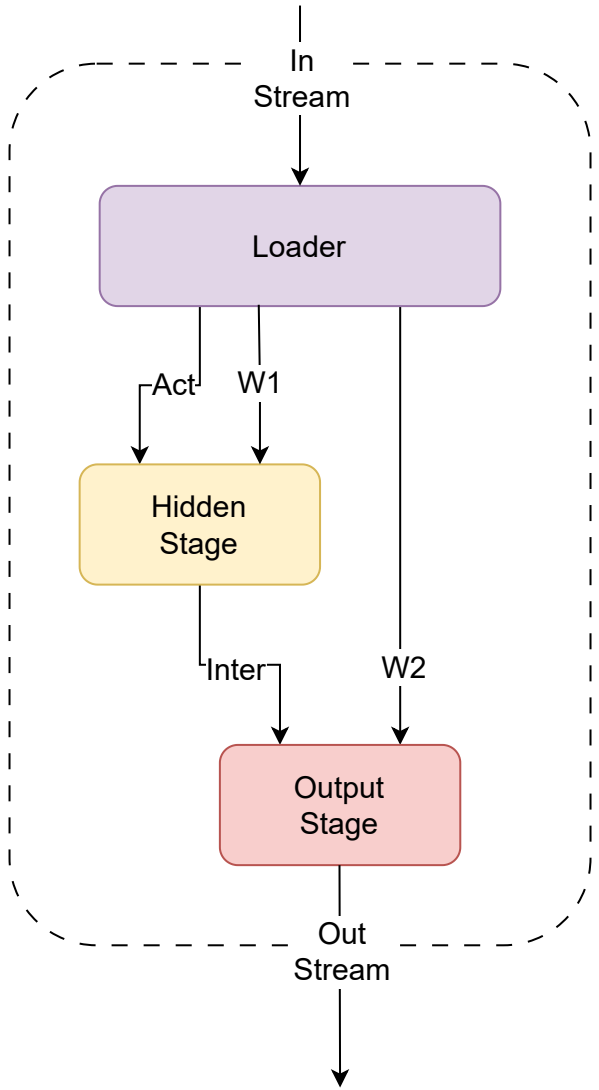
	Non-Optimized (-O0)		Optimized (-O2)	
Sub Component	**Time (s)**	**Time (%)**	**Time (s)**	**Time (%)**
QKV Gen.	30.7	25.8	5.1	18.3
Swin Attention	11.3	9.5	2.0	7.2
FFN	66.1	55.5	18.8	67.6
Others	11.1	9.3	1.9	6.8
Total	119.2	100	27.8	100

The quantized **FFN** optimized routine takes around **68%** of the execution time $\rightarrow$ conversions between `int` and `float` benefit little from opt.

FFN Accelerator - Architecture Topview

The pipelined architecture consists of:

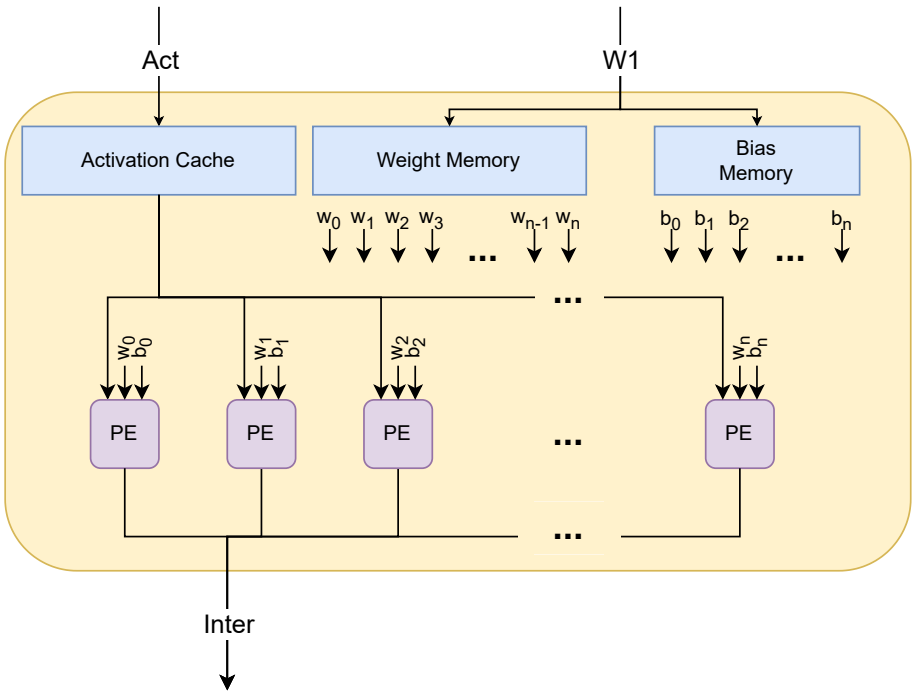
- **The Loader** → Separates the input data into 3 distinct FIFOs, avoiding HLS dataflow violations
- **Hidden Stage** → Performs the first linear transformation and Hardswish4 activation
- **Output Stage** → Performs the second linear transformation

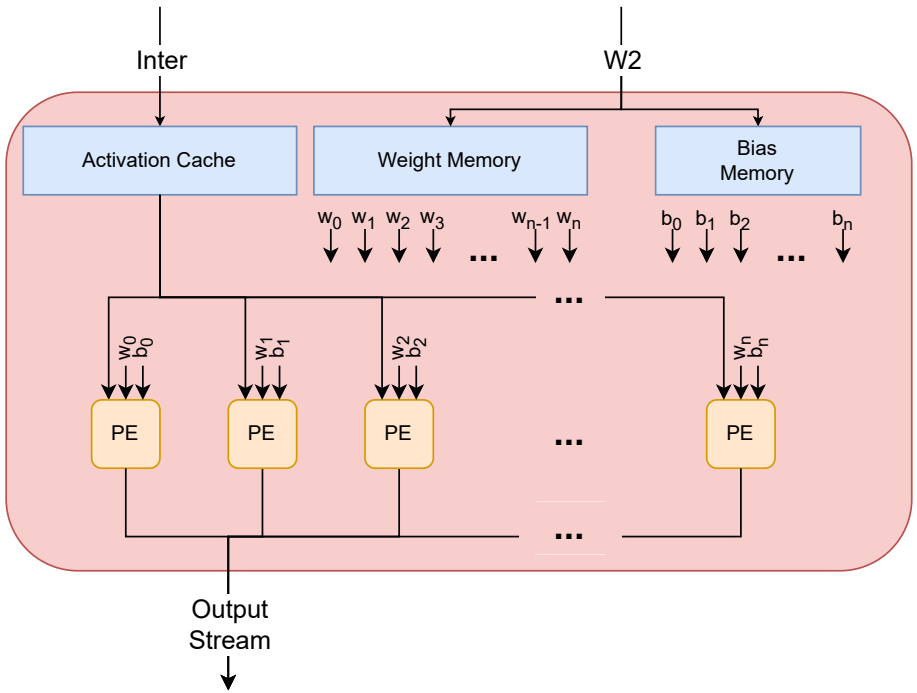


Functional Stages

Both stages were designed with a similiar structure:

- **2 memory units** $\rightarrow$ store the **complete** bias and weight maps
- **1 cache** $\rightarrow$ stores current activation **row in use**.
- **8 PE array** $\rightarrow$ parallel computation of output elements

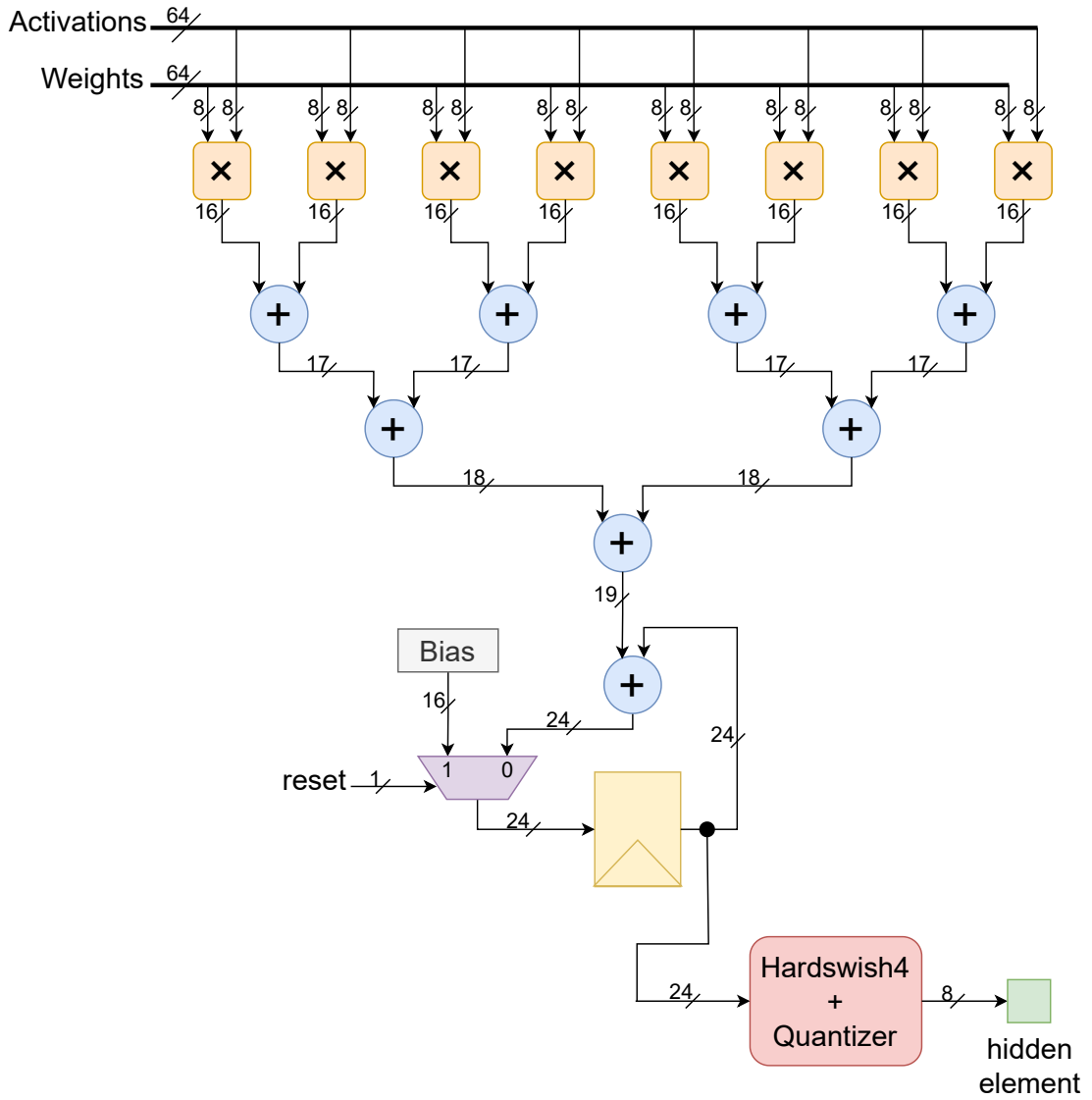




PEs Design

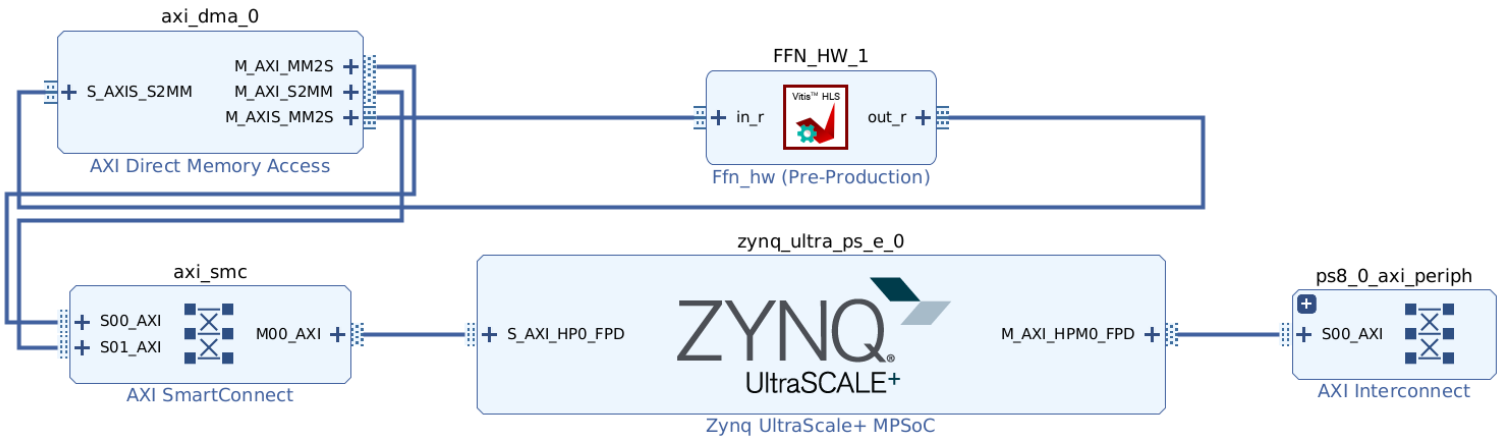
Both PE designs contain:

- **8** parallel **multipliers**
- **1 adder tree**
- The **hidden** stage **accumulator** has 24-bits
- A `Hardswish4` **computing unit**, follows the hidden stage accumulator
- The **output** stage **accumulator** has 26-bits



ARES Hardware/Software System

The FFN accelerator was integrated with the embedded Processing System to form the **ARES** (**A**cele**R**ated **E**mbedded **S**win) system



ARES Hardware/Software System - Results

	Metric	ARES (150 MHz)
Perf.	Setup Slack	0.27 ns
	FFN Speed-up	313.33×
	Total Speed-up (Amdahl's ceiling = 3.1×)	2.34×
Power	Sw-only On-Chip Total	3.33 W
	Hw/Sw On-Chip Total	3.56 W
	Hw/Sw + DDR + VCCINT	8 W
Area	LUTs Used	17,071
	FFs Used	10,479
	DSPs Used	8
	36 Kb BRAMs Used	36

ARES Hardware/Software System - Results

	Metric	ARES (150 MHz)
Perf.	Setup Slack	0.27 ns
	FFN Speed-up	313.33×
	Total Speed-up (Amdahl's ceiling = 3.1×)	2.34×
Power	Sw-only On-Chip Total	3.33 W
	Hw/Sw On-Chip Total	3.56 W
	Hw/Sw + DDR + VCCINT	8 W
Area	LUTs Used	17,071
	FFs Used	10,479
	DSPs Used	8
	36 Kb BRAMs Used	36

Conclusions

- Developed **Hardware-RialFormer**, a hardware-efficient Swin-based model.
- Developed a extendable **embedded C Swin Layer routine**, for the first-time **open-source**
- Designed a scalable and modifiable **FFN accelerator** in HLS
- Implemented an **FFN accelerator** which **speeds-up** its execution in 313$\times$
- Demonstrated **ARES**, a fully functional Hw/Sw system for a full Swin Layer execution

Future Work

The **ARES** system has great expansion potential:

- Design the Decoder Stack and CNN Stem with C CNN routines
- Use alternative **quantization methods** for possible better performance
- Extend the HLS design to use in more Swin Blocks
- Add a **simpler MatMul accelerator** for increased acceleration.

THANK YOU!

Questions?

Mean Intersection over Union

measures the overlap between the predicted segmentation mask and the ground truth mask. The mIoU is calculated as:

$$\text{mIoU} = \frac{1}{N} \sum_{i=1}^{N} \frac{TP_i}{TP_i + FN_i + FP_i} \tag{1}$$

where N is the total number of classes and i is the number of pixels in the image.

Mean F1 score

Mean F1 score (mF1) is a metric that combines $precision$ and $recall$ metrics to provide a single value for evaluating classification performance. mF1 is defined as:

$$\mathrm{mF1} = \frac{1}{N} \sum_{i=1}^{N} F1_i \tag{2}$$

where $F1$ is defined for each class i as:

$$\mathrm{F1} = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{3}$$

Mean F1 score

with $precision$ and $recall$ calculated as:

$$\text{precision} = \frac{TP}{TP + FP} \tag{4}$$

$$\text{recall} = \frac{TP}{TP + FN} \tag{5}$$

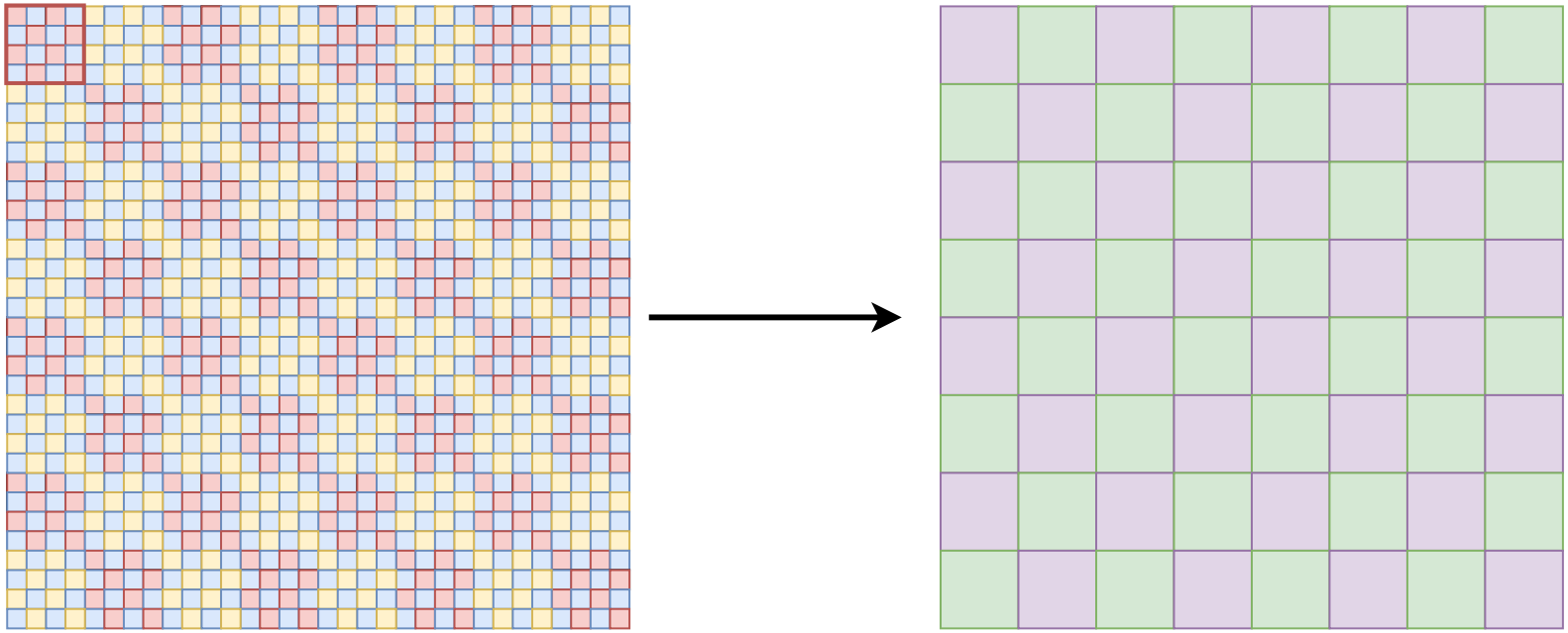
$precision$ measures the accuracy of positive predictions while $recall$ measures the model's ability to find all relevant instances, this is the model's sensitivity

Overall Accuracy

Overall Accuracy (OA) evaluates the percentage of correct predictions in the total prediction space of N classes, as

$$\mathrm{OA} = \frac{1}{N} \sum_{i=1}^{N} \frac{TP_i + TN_i}{TP_i + FP_i + TN_i + FN_i} \tag{6}$$

Input processing - Tokenization



Slices of the input are converted into condensed, coarser-grain representations, named tokens (or patches in ViTs).

Attention - Attention Mechanism

The **complexity** of the attention algorithm for **MSA** is:

$$\mathcal{O}(\mathrm{MSA}) = N^2 \cdot d$$

$N \rightarrow$ number of tokens

$d \rightarrow$ dimensions of the input.

In **WSA** the **complexity** is given by:

$$\mathcal{O}(\mathrm{WSA}) = M^2 \cdot N \cdot d,$$

$M \rightarrow$ number of tokens per window

$N \rightarrow$ total number of tokens

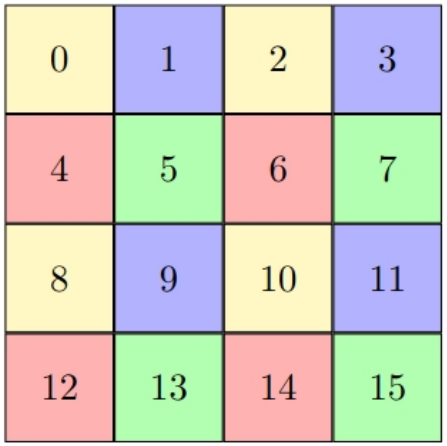
$d \rightarrow$ dimensions of the input.

Output processing

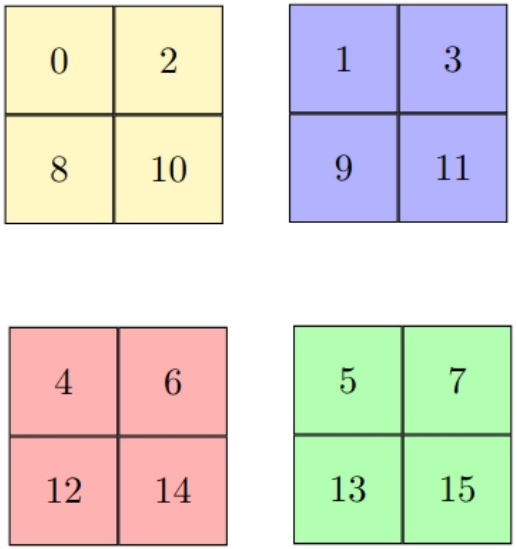
The output processing component is task-specific, ranging from:

- **Simple MLPs** $\rightarrow$ classification
- **CNNs or DNNs** $\rightarrow$ object detection, image segmentation
- **Transformer-based** $\rightarrow$ object detection, image segmentation

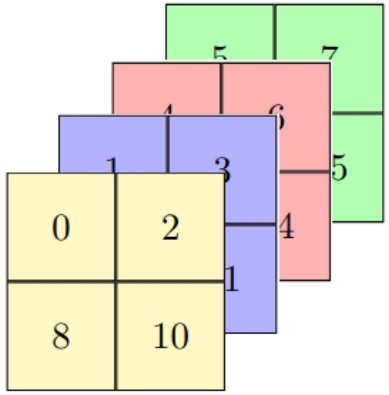
Patch Merge Operation?



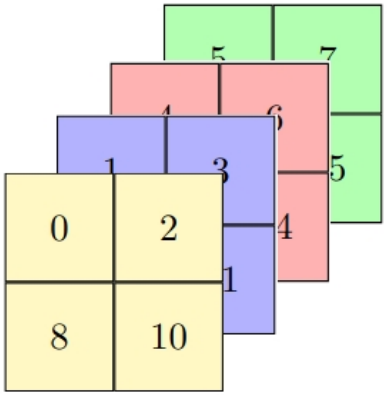
(1)

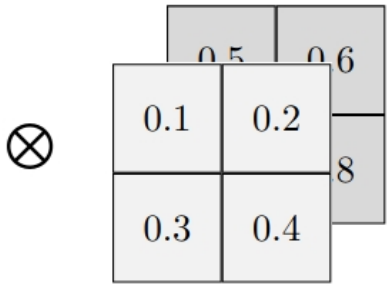


(2)

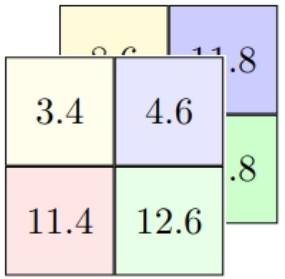


(3)





(4)



(5)

GELU Replacement

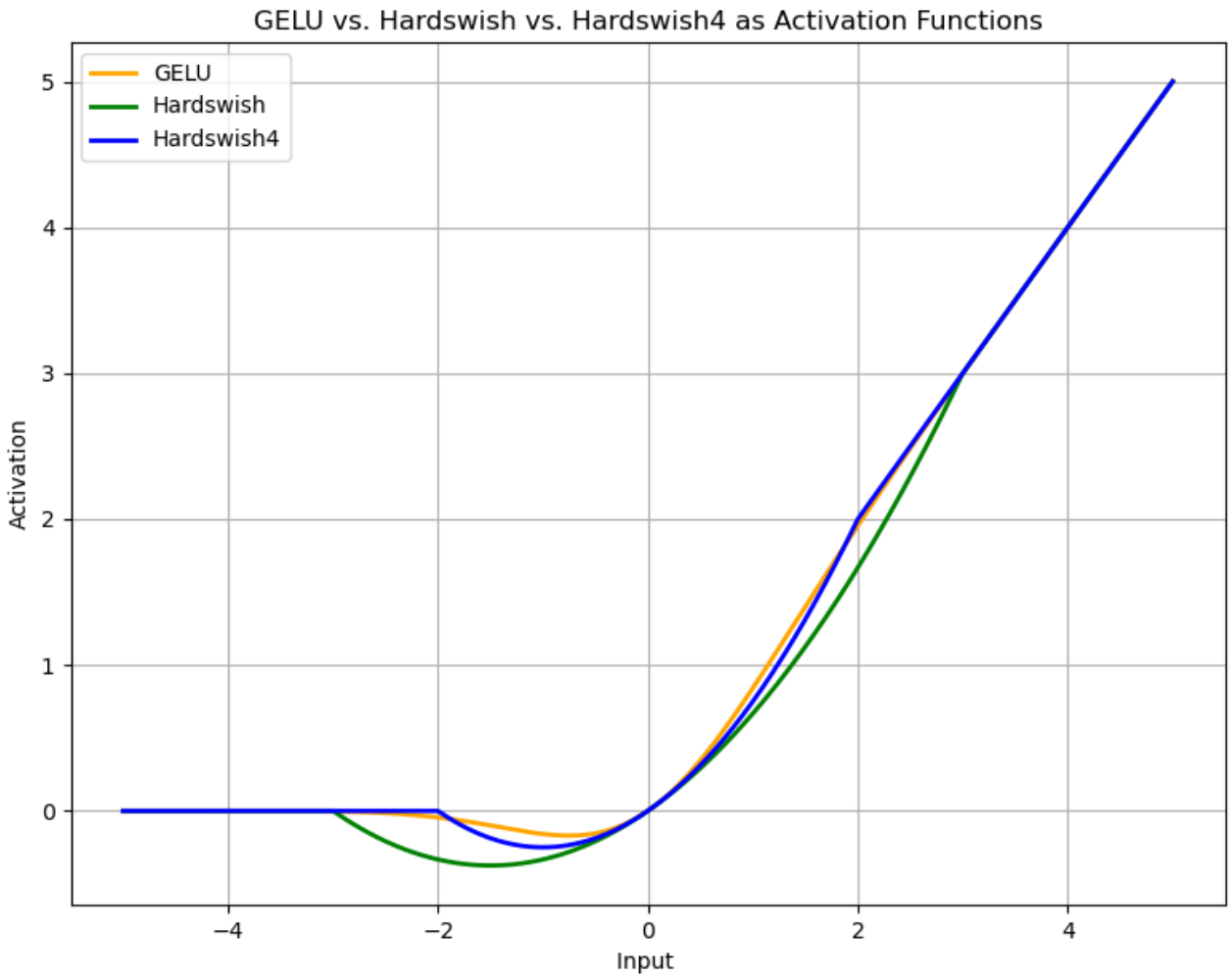
An alternative to GELU is the **Hardswish** function, based on the RELU function.

$$\text{Hardswish}(x) = x \cdot \frac{\text{ReLU6}(x+3)}{6}$$

A modified version of Hardswish, **with a base two denominator** - **Hardswish4**, is also a possible alternative

$$\text{Hardswish4}(x) = x \cdot \frac{\text{ReLU4}(x+2)}{4}$$

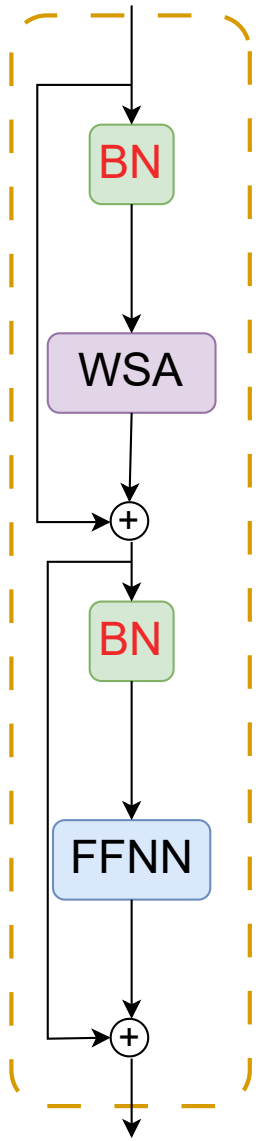
GELU Replacement - Form function comparison



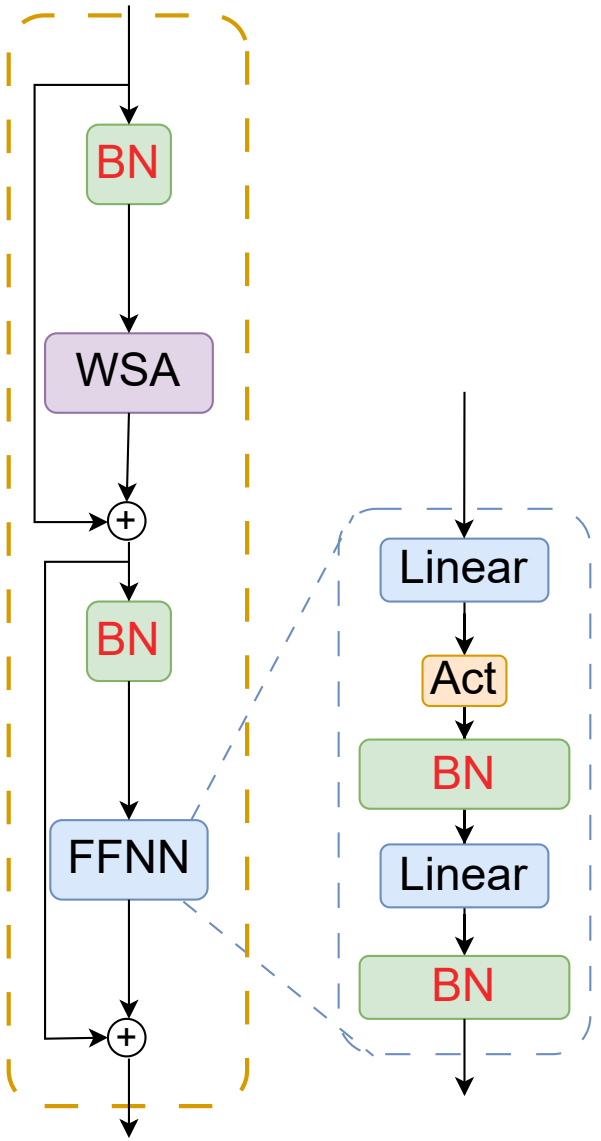
LayerNorm Replacement

Two normalization replacement strategies, based on channel-wise BatchNorm, were tested and evaluated:

- **SimpleNorm**
- **FFNNorm**.



SimpleNorm



FFNNNorm

Softmax Replacement

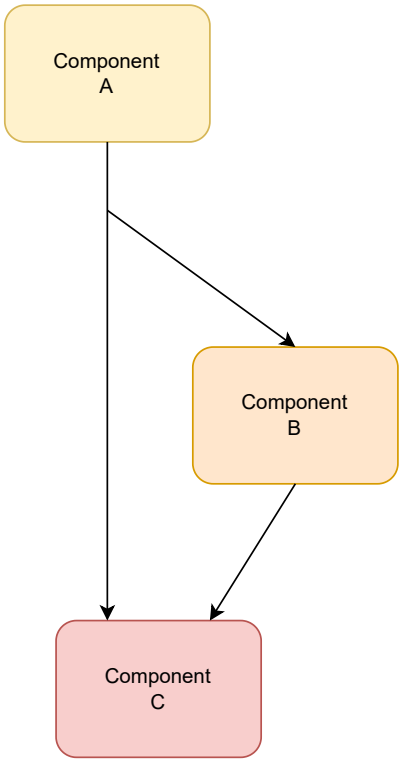
SquareMax $\rightarrow$ **hardware-friendly** alternative to Softmax, preserving percentage distribution.

$$\text{Squaremax}(x_i) = \frac{\text{ReLU}(x_i)^2}{\sum_j \text{ReLU}(x_j)^2 + \epsilon}$$

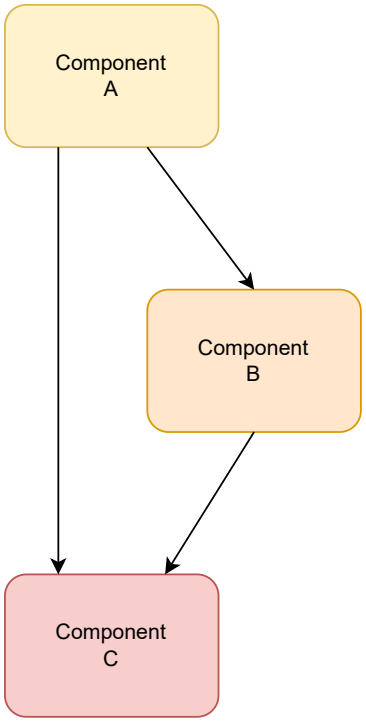
Problem $\rightarrow$ **loses** information on **negative values** (clamping them to zero), which can be **critical** when the Transformer model is **pre-trained** as is the case with the AerialFormer-T.

HLS Dataflow Violations?

The HLS dataflow pragma has to follow the **single-consumer per-producer** rule:



Dataflow violation



Corrected design

Why is the max speedup 3.1% ?

Answer $\rightarrow$ **Amdahl's Law**

In the quantized, embedded implementation the FFN takes around **67.6%** of the execution time.

By accelarating the FFN to the **limit** where **it would take 0.0%**, the new execution delay would be 32.4%:

$$\text{Speed-up} = \frac{\text{Baseline}}{\text{Accelerated}} = \frac{100}{32.4} = 3.1$$

Minimum FPGA resources?

The minimum BRAM needed to expand the proposed design is 6.30 Mb, assuming the current 50% utilization of the mapped BRAMs.

	Stage 1 size (Mb)	Stage 2 size (Mb)	Stage 3 size (Mb)	Stage 4 size (Mb)
Act.	12.60	6.29	3.15	1.50
Weigthts	0.59	2.4	9.2	36.80

With the proposed MatMul accelerator expansion the weights and activations can be switched in the FFN, ensuring the smaller matrix is kept in BRAM.