Modeling the Vision Transformer in SystemC

Aniket Fasate
College of Engineering
Northeastern University
Boston, MA 02115
fasate.a@northeastern.edu

Baibhav Kumar Pathak
College of Engineering
Northeastern University
Boston, MA 02115
pathak.ba@northeastern.edu

Luyue Zhang
College of Engineering
Northeastern University
Boston, MA 02115
zhang.luyue@northeastern.edu

Robert D'Antonio
College of Engineering
Northeastern University
Boston, MA 02115
dantonio.ro@northeastern.edu

Tilak Marupilla
College of Engineering
Northeastern University
Boston, MA 02115
marupilla.t@northeastern.edu

***Abstract*—The Vision Transformer (ViT) project explores the application of transformer-based architectures in the domain of image classification, leveraging the computational power of SystemC for hardware-software co-simulation. This project focuses on implementing core components of a ViT model, including Patch Embedding, Transformer Encoder, and Classification Blocks, while supporting pre-trained models in a standardized format. By utilizing modular design principles, the system enables flexible integration, testing, and scalability.**

The testbench is designed to rigorously evaluate the model's functionality and performance by dynamically loading pre-trained weights, processing test images, and measuring critical metrics such as inference time and classification accuracy. Each module is implemented as an independent SystemC block, simulating essential transformer operations such as self-attention, feed-forward refinement, and residual connections.

This implementation demonstrates the feasibility of high-level design for modern deep learning architectures in SystemC, bridging the gap between software flexibility and hardware-level simulation. The resulting system serves as a robust platform for exploring transformer models in hardware, optimizing for both performance and adaptability in real-world scenarios.

I. Introduction

A. Motivation

The Vision Transformer (ViT) architecture has redefined the field of computer vision by adapting the transformer model to process images. Unlike convolutional neural networks (CNNs), ViTs divide images into fixed-size patches and treat them as sequences, enabling the application of self-attention mechanisms to capture global relationships among patches[1]. This architecture has achieved state-of-the-art performance in various image classification tasks, demonstrating scalability and effectiveness when trained on large datasets[2][3]. However, deploying ViTs on hardware-constrained devices remains a challenge due to their high computational and memory requirements[4]. This motivates the exploration of hardware-aware design methodologies that enable efficient implementation and testing of ViTs in hardware platforms.

B. Problem Statement

Despite their performance advantages, Vision Transformers face significant barriers to deployment in hardware systems:

1) **Computational Intensity:** The self-attention mechanism in ViTs involves extensive matrix multiplications, leading to high computational complexity and power consumption[5][6].
2) **Memory Footprint:** The large number of parameters and intermediate data requires substantial memory, complicating deployment on devices with limited resources[7].
3) **Hardware Integration:** Efficiently mapping ViT architectures onto hardware platforms, such as FPGAs and ASICs, necessitates custom design approaches that align with their unique computational patterns[8].

These challenges highlight the need for hardware-software co-design techniques to optimize ViT implementations for practical applications.

C. Solution Outline

This project implements a Vision Transformer using **SystemC**, a system-level modeling language that enables hardware-software co-design and simulation. The implementation includes the following key components:

1) **Patch Embedding:** The input image is divided into patches, which are flattened and embedded into a vector representation suitable for processing by the transformer encoder.
2) **Transformer Encoder:** A series of layers utilizing self-attention and feed-forward mechanisms to extract global features from image patches.
3) **Classification Head:** A fully connected network that maps the extracted features to class probabilities for image classification.

To ensure functionality and performance, a comprehensive testbench is developed to:

- Load pre-trained weights from the original Vision Transformer model[1] for modular components (patch embedding, encoder, classification head).
- Simulate inference on test images.
- Evaluate metrics such as inference time, classification accuracy, and hardware efficiency.

This modular and flexible approach provides a platform for assessing the feasibility and performance of ViT implementations in hardware environments.

D. Benifits

The proposed SystemC-based Vision Transformer implementation offers several advantages:

1) **Hardware-Software Co-Design:** Enables the integration of software-level machine learning algorithms with hardware-level implementations, optimizing for resource constraints.
2) **Scalability:** The modular design supports various Vision Transformer configurations and facilitates experimentation with pre-trained models.
3) **Performance Evaluation:** Allows systematic benchmarking of inference time, memory usage, and accuracy, guiding design optimizations for hardware efficiency.
4) **Flexibility:** Supports testing with multiple pre-trained models and datasets, making the system adaptable to different use cases.

By addressing these challenges, this work bridges the gap between high-level Vision Transformer models and their practical deployment on hardware, contributing to the broader goal of efficient machine learning in resource-constrained environments.

II. Related Work

A. Vision Transformers for Image Recognition

The Vision Transformer (ViT) architecture, introduced by Dosovitskiy et al., marked a paradigm shift in computer vision by replacing convolutional operations with self-attention mechanisms to process sequences of image patches[1]. This approach enables ViTs to achieve state-of-the-art performance in image classification tasks while scaling effectively with larger datasets. Subsequent works, such as the Swin Transformer, extended this concept by introducing hierarchical processing to improve efficiency in capturing local and global patterns[2].

B. Algorithmic Optimization for ViTs

To address the computational challenges posed by ViTs, researchers have proposed various optimization techniques. Du et al. provided a comprehensive survey on techniques like quantization, pruning, and knowledge distillation to make ViTs more hardware-friendly[4]. Wang and Chang proposed a row-wise accelerator for ViTs, restructuring self-attention into dot-product primitives to improve computational efficiency on hardware platforms[5].

C. Hardware-Software Co-Design for ViTs

Efforts to integrate ViTs into hardware systems have led to the development of frameworks like CHOSEN, which automates the generation of ViT accelerators for FPGA devices by transforming high-level models into efficient hardware implementations[8]. Similarly, Zhang et al. introduced ViTA, a Vision Transformer Accelerator tailored for edge computing, leveraging pipeline optimizations to achieve nearly 90% hardware utilization[7].

D. SystemC-Based ViT Implementation

Despite advancements in hardware-aware ViT designs, the use of SystemC for simulating ViTs in hardware-software co-design environments remains underexplored. SystemC offers a robust platform for evaluating the feasibility of ViTs in hardware systems, bridging the gap between software-level modeling and hardware implementation. This project addresses this gap by implementing a ViT in SystemC, focusing on modular components like patch embedding, transformer encoders, and classification blocks. The inclusion of a testbench allows for comprehensive validation and performance evaluation, paving the way for efficient ViT deployment in hardware environments.

III. Design and Implementation

We implement our solution using SystemC[15]. We chose SystemC since it allows us to assemble a model of the hardware while also enjoying the simplicity and more rapid development times that come from the software world. One of SystemC's primary functionalities is the `SC_MODULE`, which can be thought of as the hardware version of a C++ class. In our implementation, we use the `SC_MODULE` to model a specific hardware block. The `SC_MODULE`s are connected with a mix of SystemC signals (analogous to wires in the hardware realm) and C++ class constructs. When assembled together, we have three `SC_MODULE`s to represent embedding, transformer, and MLP block of the Vision Transformer. We then add a fourth `SC_MODULE` that connects these three submodules together. We call this the top-level module.

A. Embedding Block

1) Patch + Position Embedding: Implementation and Optimization: The **Patch + Position Embedding** module is a critical component in the Vision Transformer (ViT) architecture, transforming raw image data into sequences of embeddings enriched with positional information. This subsection elaborates on its design and SystemC-based implementation, emphasizing modularity, efficiency, and seamless integration with downstream components.

The Role of Patch + Position Embedding in Vision Transformers

The embedding block bridges raw image data and the transformer layers by preparing a sequence of patch embeddings. Its main responsibilities are:

- **Patch Extraction**: Divides the input image into fixed-size non-overlapping patches (e.g., 16x16 pixels), enabling spatial relationships to be preserved at the patch level. Each patch acts as a localized representation of the image, forming the fundamental unit of computation for the Vision Transformer.
- **Flattening and Embedding**: Flattens each patch into a 1D vector to convert 2D spatial information into a

format suitable for linear operations. Maps the flattened vector into a high-dimensional space (e.g., 768 dimensions) using a learnable projection matrix, allowing the transformer to process features in a unified embedding space.

- **Positional Encoding**: Adds learnable positional embeddings to each patch embedding to incorporate spatial context, ensuring that the transformer retains information about the position of each patch within the image. Positional encoding enables the self-attention mechanism to understand spatial relationships among patches.
- **Class Token Addition**: Introduces a special `[CLS]` token, a learnable vector prepended to the sequence of embeddings, which aggregates global information for the entire image and is used for final classification.

Design Considerations for Patch + Position Embedding

a) Memory Efficiency: Efficient memory usage is essential when processing large images or handling high-dimensional embeddings:

- Intermediate data, such as image patches and embeddings, are temporarily stored in on-chip memory buffers to reduce latency and memory bandwidth bottlenecks.
- Large weight matrices used for projection are loaded dynamically from external memory to conserve on-chip resources.

b) Scalability: The embedding block is designed to handle a range of configurations to accommodate diverse applications:

- **Image Resolutions**: Flexible resizing options allow input images of various resolutions to be scaled to a standard size (e.g., 224x224).
- **Patch Sizes**: Configurable patch dimensions (e.g., 8x8, 16x16, or 32x32) enable experimentation with finer or coarser granularities.
- **Embedding Dimensions**: Adjustable embedding sizes (e.g., 512, 768, or 1024) allow adaptation to different Vision Transformer architectures.

c) Integration with Transformer Encoder: Seamless integration with the transformer encoder requires:

- Ensuring the output dimensions of the embedding block align with the input requirements of the encoder.
- Standardizing data formats to simplify communication between modules.

SystemC-Based Implementation of Patch + Position Embedding

d) Module Structure: The embedding block is implemented as a standalone **SystemC** module to ensure modularity, scalability, and reusability.

Input Ports:

- `img_path`: The file path of the input image to be processed.
- `weights_dir`: Directory containing pre-trained weights for the projection matrix and bias.
- `start`: A control signal to initiate the embedding process.

Output Ports:

- `embedding_sequence`: A 2D array (`sc_out<sc_fixed<16,8>>`) containing the final sequence of position-encoded embeddings.
- `done`: A signal that indicates the embedding process has been completed.

Internal Components:

- Buffers for temporarily storing image patches, intermediate embeddings, and positional encodings.
- Dynamic memory structures to handle varying patch sizes and embedding dimensions.

e) Dataflow and Computation:

1) **Image Preparation**:
   - **Loading**: Reads the input image using OpenCV and converts it to an RGB format if not already in this format.
   - **Resizing**: Scales the image to a standard resolution (e.g., 224x224 pixels) to ensure uniformity across different input images.
   - **Normalization**: Normalizes pixel values to a [0, 1] range and optionally standardizes them based on the mean and standard deviation of the training dataset.
2) **Patch Extraction**:
   - **Sliding Window**: Divides the resized image into fixed-size patches using a sliding window with a stride equal to the patch size.
   - **Temporary Storage**: Stores extracted patches in a local buffer to optimize memory access patterns during subsequent operations.
3) **Linear Projection**:
   - **Weight Matrix Multiplication**: Each patch is flattened into a vector and multiplied by a learnable projection matrix, mapping it to the embedding dimension (e.g., 768).
   - **Bias Addition**: A learnable bias vector is added to each embedding, introducing additional flexibility for fine-tuning.
4) **Class Token and Positional Encoding**:
   - **Class Token**: A special learnable vector is prepended to the sequence of patch embeddings.
   - **Positional Encoding**: Adds a unique learnable positional vector to each patch embedding, preserving spatial relationships in the sequence.
5) **Output Transmission**:
   - **Buffering**: The final sequence of position-encoded embeddings is stored in the `embedding_sequence` output port.
   - **Completion Signal**: Asserts the `done` signal to indicate that the embedding process is finished.

f) Optimization Strategies:

- **Memory Optimization**:
  - On-chip buffers reduce latency and minimize memory bandwidth bottlenecks.

  - Data is cached in local registers during computations.
- **Parallel Processing**:
  - Parallel computation of embeddings for multiple patches accelerates the overall embedding process.
  - Efficient multi-threading techniques simulate parallelism during SystemC execution.
- **Dynamic Parameter Loading**:
  - Pre-trained weights and biases are stored as CSV files, which are parsed and loaded at runtime, allowing flexibility in testing different models.

g) Verification and Validation:

- **Functional Testing**:
  - Compare outputs with a PyTorch-based reference implementation to ensure functional equivalence.
  - Validate outputs for various configurations of patch sizes, embedding dimensions, and positional encodings.
- **Stress Testing**:
  - Test with high-resolution images and large patch sizes to ensure the module handles edge cases without performance degradation.
- **Performance Metrics**:
  - Measure the latency for embedding generation under varying image resolutions and patch dimensions.
  - Evaluate memory usage and ensure optimal resource allocation.

Workflow Overview

Input:

- **Image File**: Path to the image to be processed.
- **Weights Directory**: Directory containing projection matrix and bias in CSV format.

Processing:

1) Load, resize, and normalize the input image.
2) Extract patches and compute embeddings using pre-trained weights.
3) Add positional encodings and the class token.

Output:

- A sequence of embeddings stored in the `embedding_sequence` buffer, ready for the transformer encoder.

B. Transformer Block

The positioned embedding vector is then passed to the transformer block, and processed by the encoder layer for 12 times. The encoder architecture is implemented in the encoder.cpp file in the src/sw/ folder. It is sensitive to the "run_encoder_signal", takes "input_tensor" as the input data, and generates the output data "output_tensor" for use by subsequent layers. The input tensor and output tensor has a dimension of 197x768. Dropout has been deleted from the code. (A lesson learned here is that when a group is cooperating on a task, it is better to start analyze the task from the top down. Thus, the structure of the system will be discussed first as well as the connection of each individual work. Mistake I made such as adding dropout to the inference code could therefore be avoided). The encoder layer is wrapped as a SC_MODULE with the following functions :

- **Initialize Weights:** The scheduler file can use this function to load weights to the encoder file by running: encoder.initialize_weights(encoder_weights);
- **Layer Normalization:** LayerNorm is used to scale the data as a standard normal distribution for more stable training. It does not change the dimension of the tensor.
- **Multi-Head-Attention Mechanism:** This function feed the feature tensor into 12 heads, computing the attention score in their own space. The K(key), Q(query), V(value) matrix of each head has a dimension of 197x64. After concatenate the results from each head, the dimension of the tensor is restored to 197x768.
- **MLP block:** The multi-layer perceptron block has two fully connected layers. The feature tensor will be expanded by the first layer, scaling up the dimension 4 times larger. After applying GELU, the features are projected back to the original dimension by the second layer.
- **Residual Connection:** This function adds the input values to the output values to better pass the features to the subsequent layers.

When the run_encoder_signal is positive, the SC_THREAD process will be activated, and hence start processing the input tensor and writes the output to the sc_out signal. The following code snippet demonstrates the running process of one encoder layer.

```
// running encoder
void run_encoder_process() {
    while (true) {
        wait(run_encoder_signal.posedge());

        tensor = input_tensor.read();
        layer_norm(tensor);
        multi_head_attention(tensor);
        tensor = residual_connection(
input_tensor, tensor);
        layer_norm(tensor);
        mlp_block(tensor);

        output_tensor.write(tensor);
    }
}

SC_CTOR(Encoder) {
    SC_THREAD(run_encoder_process);
    sensitive << run_encoder_signal;
}
```

C. MLP Classifier

The **MLP Classifier** serves as the final stage in the Vision Transformer (ViT) architecture, responsible for translating refined feature embeddings from the Transformer Encoder into actionable class predictions. This subsection details the design, implementation, and optimization of the MLP Classifier, emphasizing its critical role in image classification tasks.

1) The Role of the MLP Classifier in Vision Transformers: The MLP Classifier operates on the refined feature representations generated by the Transformer Encoder. Its key responsibilities are:

- **Input Processing (Extraction of `[CLS]` Token)**:
  - Extracts the `[CLS]` token, which aggregates global information about the input image, from the encoded sequence.
  - The `[CLS]` token is a high-dimensional vector (e.g., 768 dimensions) that represents the entire image in feature space.
- **Linear Transformation (First Layer)**:
  - Projects the extracted `[CLS]` token into a higher-dimensional space (e.g., 3072 dimensions) to enhance the model's ability to capture complex patterns.
  - Adds a learnable bias vector to the projection, introducing flexibility for feature fine-tuning.
- **Non-Linear Activation (GELU)**:
  - Applies the GELU activation function to introduce non-linearity, enabling the model to learn intricate patterns in the data.
  - GELU provides a smoother and more accurate approximation of ReLU, ensuring better gradient flow during training.
- **Final Linear Transformation (Second Layer)**:
  - Reduces the dimensionality of the activated vector to match the number of target classes (e.g., 1000 for ImageNet).
  - Produces logits for each class, representing the unnormalized probabilities of the input belonging to each class.
- **Optional Step - Softmax Activation**:
  - Converts logits into normalized probabilities, facilitating interpretability during inference.
  - Softmax is applied only during testing or evaluation phases and not during the training process.

2) Design Considerations for the MLP Classifier:

a) Arithmetic Precision: Balancing computational accuracy and hardware efficiency is critical:

- Floating-point operations provide high precision but demand more resources, making them suitable for initial validation.
- Fixed-point arithmetic is considered for deployment scenarios, reducing power and area requirements while maintaining acceptable accuracy.

b) Memory Management: Efficient handling of large weight matrices and intermediate tensors is essential:

- On-chip buffers are used to store weights and intermediate results, minimizing latency.
- Dynamic loading of weights from external memory enables flexibility for different models and configurations.

c) Parallelism and Scalability: The design ensures high throughput and adaptability:

- Implements pipelining for the sequential stages of computation to overlap operations and reduce latency.
- Supports varying embedding dimensions and numbers of output classes, ensuring scalability across tasks and datasets.

3) SystemC-Based Implementation of the MLP Classifier:

a) Module Structure: The MLP Classifier is implemented as an independent **SystemC** module, ensuring modularity and reusability.

Input Ports:

- `cls_token`: A vector of `sc_in<sc_fixed<16,8>>` representing the `[CLS]` token embedding.
- `start`: Control signal to initiate the classification process.
- `reset`: Signal to reset the internal state of the module.

Output Ports:

- `classification`: A vector of `sc_out<sc_fixed<16,8>>` representing the final classification logits.
- `done`: Signal indicating the completion of the classification process.

Internal Components:

- Buffers for storing weight matrices (`W1`, `W2`) and bias vectors (`b1`, `b2`).
- Registers for intermediate computations, such as the activated vector and final logits.

b) Dataflow and Computation:

1) **Weight Initialization**:
   - Loads weights (`W1`, `W2`) and biases (`b1`, `b2`) dynamically from CSV files during initialization.
   - Verifies the integrity of loaded weights to prevent errors during computation.
2) **Input Extraction**:
   - Reads the `[CLS]` token from the input port and stores it in a local register.
3) **First Linear Transformation**:
   - Multiplies the `[CLS]` token vector with the weight matrix (`W1`).
   - Adds the bias vector (`b1`) to compute the intermediate vector (`z`).
4) **Non-Linear Activation**:
   - Applies the GELU activation function element-wise to the intermediate vector (`z`).
5) **Second Linear Transformation**:
   - Multiplies the activated vector (`a`) with the second weight matrix (`W2`).
   - Adds the second bias vector (`b2`) to produce the final logits (`y`).
6) **Output Transmission**:

- Writes the final logits to the `classification` output port.
- Asserts the `done` signal to indicate completion.

c) Optimization Strategies:

- **Fixed-Point Arithmetic**:
  - Reduces the bit-width of computations (e.g., `sc_fixed<16,8>`) to lower resource usage and power consumption.
- **Parallel Execution**:
  - Parallelizes matrix-vector multiplications in linear layers to enhance throughput.
- **Memory Optimization**:
  - Implements on-chip buffers for intermediate computations, reducing external memory accesses.

d) Verification and Validation:

- **Functional Testing**:
  - Validates output logits against a reference PyTorch implementation to ensure correctness.
- **Stress Testing**:
  - Tests with large numbers of classes and varying input dimensions to evaluate scalability.
- **Performance Analysis**:
  - Measures latency, memory usage, and accuracy under different configurations.

4) Workflow Overview: **Input**:

- `[CLS]` token vector from the Transformer Encoder.
- Weight and bias files for pre-trained MLP layers.

Processing:

1) Load pre-trained weights and biases.
2) Compute intermediate and final logits through linear transformations and activation.
3) Transmit logits to the output.

Output:

- Final classification logits stored in the `classification` port.
- `Done` signal asserted to indicate completion.

D. Top Level Module

The top-level module encapsulates the embedding block, the MLP classifier, and twelve transformer blocks. In addition to wiring the blocks together, the top level contains substantial control logic to ensure the blocks are scheduled in such a way that data and weights are always available. We divide the control logic into three categories. SystemC signals, C++ constructs, and Weight Loading.

1) SystemC Signals: We aim to use SystemC signals as much as possible, as they are directly parallel to the wires found in the hardware. SystemC signals are used for passing basic parameters to the modules and for the scheduling signals. Each submodule is connected to a start and done signal. When its start signal is set to high from the top level module, the submodule begins executing. The submodule sets the done signal upon execution completion, at which point the top level module will send the start signal to the next submodule.

2) C++ Constructs: Our implementation makes extensive use of standard C++ functionality, primarily to handle data. The input and output matrices of a submodule are simply stored on the heap, and the top level module passes pointers to the necessary data to each submodule. Each submodule receives a pointer to its input buffer and output buffer, so there is no simulation time wasted on deep copies.

3) Weight Loading: A directory of CSV files containing the weights and biases of the model is generated by a Python script that loads the weights from a PyTorch implementation of the pre-trained vision transformer. For convenient packaging, we include this script in the test rule of the Makefile. Once the CSV files have been created, the modules themselves are responsible for loading their own weights. Each module is linked with a weight loader header file containing functions to load weights to either a C++ Eigen Vector (for 1D weights) or a C++ Eigen Matrix (for 2D weights). This load-at-runtime implementation was necessary due to the size of the model; as float32, the weights combine to just over 1GB of data and was found to stall compile time significantly.

IV. Results and Conclusions

Our work on this project has demonstrated to us the value of system design languages. System C has made designing a complex model like this a much more approachable task than it would be had we used RTL or C++ exclusively. While we regret that we have not gotten the entire model functioning, we have achieved promising results from individual layers. Ultimately, we were set back by C++ implementation difficulties; translating the transformer architecture to C++ was not as easy as pulling an open source C++ implementation, and implementing it from scratch is quite cumbersome. Learning the different ways of handling matrices in C++ added further setbacks to our work.

A. Embedding Layer Proof of Functionality

The greatest success of our project is the embedding layer, which successfully operates as a `SC_MODULE` and gets called by the top level module. When comparing its results with the pre-trained Vision Transformer in PyTorch on the same input image, we found nearly identical results. This tells us that a continuation of this project is certainly worthwhile and that a functioning SystemC implementation is achievable.

V. Future Work

In our work, we have shown that implementing a Vision Transformer in SystemC is a valuable and doable task. However, more work is needed to reach the baseline that is getting reasonable outputs from our implementation. We believe that this baseline is not very far away from what we have presented here.

Once the baseline has been achieved, there are seemingly countless ideas to explore. If we had another month, we would like to explore:

- Different methods for connecting the sub-modules; via double-buffering, or some other means. - Different granularity

of sub-module implementations, and the scheduling+design implications of these changes - Designing a more generalized interface that allows loading of different models.

REFERENCES

[1] Dosovitskiy, A., et al. (2020). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. arXiv preprint arXiv:2010.11929. Available at: https://arxiv.org/abs/2010.11929

[2] YouTube Video (2020). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale (Paper Explained). Available at: https://www.youtube.com/watch?v=TrdevFK_am4

[3] GitHub Repository (2024). vit.cpp. Available at: https://github.com/staghado/vit.cpp

[4] Vaswani, A., et al. (2017). Attention is All You Need. Advances in Neural Information Processing Systems (NeurIPS). Available at: https://arxiv.org/abs/1706.03762

[5] Zhang, J., Zhou, Y., Lin, K., and Sun, J. (2018). ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. European Conference on Computer Vision (ECCV). Available at: https://arxiv.org/abs/1707.01083

[6] Han, S., Mao, H., and Dally, W. J. (2016). Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. International Conference on Learning Representations (ICLR). Available at: https://arxiv.org/abs/1510.00149

[7] Chen, Y., Emer, J., and Sze, V. (2015). EIE: Efficient Inference Engine on Compressed Deep Neural Network. IEEE International Solid-State Circuits Conference (ISSCC). Available at: https://arxiv.org/abs/1510.00978

[8] Zhang, C., Li, P., Guo, Y., and Sun, Y. (2021). MLP-Mixer: An all-MLP Architecture for Vision. arXiv preprint arXiv:2105.01601. Available at: https://arxiv.org/abs/2105.01601

[9] Baumgartner, S., Wolf, T., and Richards, R. (2011). SystemC Language Reference Manual. Accellera Systems Initiative. Available at: https://www.accellera.org/images/downloads/standards/systemc/SystemC_LRM_2.3.pdf

[10] Bartolucci, J., and Haupt, R. (2003). Modeling and Simulation of Digital Systems using SystemC. Wiley-Interscience.

[11] Narayanan, D., and Schneider, G. (2003). Practical SystemC. Springer Science Business Media.

[12] Wang, A., et al. (2021). Faster Transformers: Hardware-Efficient Transformer Design for Real-Time Inference. IEEE Transactions on Neural Networks and Learning Systems. Available at: https://ieeexplore.ieee.org/document/9411463

[13] Shi, W., et al. (2022). Hardware Acceleration for Vision Transformers: An Analysis of the ViT Architecture. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. Available at: https://ieeexplore.ieee.org/document/9411482

[14] King, M. L., et al. (2020). Optimizing Transformer Models for High-Performance Hardware Acceleration. ACM International Conference on Computer-Aided Design (ICCAD). Available at: https://arxiv.org/abs/2003.11212

[15] Narayanan, D., and Schneider, G. (1999). SystemC: From Virtual Register Transfer Level Models to Hardware Implementations. IEEE Design Test of Computers, vol. 16, no. 1, pp. 38-47. Available at: https://ieeexplore.ieee.org/document/752009

[16] Peng, L., et al. (2021). Accelerating Vision Transformers on FPGAs via Layer Fusion and Data Reuse. IEEE International Conference on Computer Design (ICCD). Available at: https://arxiv.org/abs/2107.01565

[17] He, K., et al. (2016). Deep Residual Learning for Image Recognition. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Available at: https://arxiv.org/abs/1512.03385

[18] Lin, T.-Y., Goyal, P., Girshick, R., He, K., and Dollar, P. (2018). Focal Loss for Dense Object Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI). Available at: https://arxiv.org/abs/1708.02002

[19] Wu, Y., Wang, C., and Qiu, T. (2021). Hardware-Efficient Transformer Accelerators for Real-Time Applications. IEEE Design Test of Computers. Available at: https://ieeexplore.ieee.org/document/9433766

[20] Chu, P. P. (2009). FPGA Prototyping by SystemVerilog Examples: Xilinx MicroBlaze MCS SoC. Wiley.