AI Hardware: FPGA-Accelerated ASL Recognition

Maïva Ndjiakou Kaptue, Wil Berling, Daniel Lee

December 2025

1 Introduction

Sign language recognition is an important human–computer interaction problem with applications in accessibility, education, and assistive technologies. Automatic interpretation of sign language requires robust perception of hand gestures under real-world conditions, including variations in lighting, background, hand shape, and motion.

Recent advances in deep learning have significantly improved the accuracy of vision-based sign language recognition systems. However, most state-of-the-art approaches rely on general-purpose CPUs or GPUs, which may not be suitable for embedded or low-power applications. This motivates the exploration of hardware-accelerated solutions capable of performing inference efficiently while maintaining acceptable accuracy.

In this project, we investigate a real-time American Sign Language (ASL) alphabet recognition system and study its deployment on an FPGA-based embedded platform. We focus on a constrained subset of ASL letters and compare a standard neural network implementation running on a personal computer with a hardware-accelerated implementation targeting the PYNQ-Z1 board. The goal is to analyze the trade-offs between flexibility, performance, and energy efficiency in software versus hardware-based inference.

2 Problem Definition and Motivation

The objective of this project is to design and evaluate a real-time ASL alphabet recognition system capable of interpreting hand gestures from a live camera stream. Given an input video frame, the system must detect the presence of a hand, extract relevant spatial features, and classify the gesture into a predefined set of ASL letters or reject it as unknown.

From a hardware perspective, deploying such a system presents several challenges. Neural networks for vision tasks often involve computationally expensive operations, such as convolutions and matrix multiplications, which can lead to high latency and power consumption when executed on a CPU. While GPUs offer significant acceleration, they are not always available or practical in embedded environments.

The motivation for this work is therefore twofold. First, we aim to build a functional and interpretable ASL recognition pipeline that operates in real time. Second, we seek to explore the benefits of FPGA-based acceleration by mapping parts of the inference workload onto programmable logic, with the PYNQ-Z1 board serving as the target hardware platform.

3 Dataset and Preprocessing

The dataset used in this project is the American Sign Language (ASL) Alphabet dataset available on Kaggle. The dataset contains grayscale images of static hand gestures corresponding to the letters of the English alphabet. Each image represents a single hand gesture captured under varying conditions.

The original images are provided at a resolution of 28×28 pixels and contain a single channel. During preprocessing, the images are normalized and reshaped to ensure compatibility with the neural network input format. The dataset is split into training and evaluation subsets to enable supervised learning and performance assessment.

During inference, a real-time preprocessing pipeline is applied to frames captured from a webcam. Hand detection is performed to localize the region of interest, which is then cropped and resized before being passed to the neural network classifier. This preprocessing pipeline is shared between the PC-based implementation and the FPGA-targeted implementation to ensure a fair comparison.

4 Neural Network Model Architecture

The ASL gesture classifier is implemented as a convolutional neural network (CNN) designed for low-resolution grayscale image classification. The architecture, exported to ONNX format and visualized using Netron, is summarized as follows.

The network input consists of a $28 \times 28 \times 1$ grayscale image. The first stage of the network applies a convolutional layer with 32 filters of size 3×3, followed by a ReLU activation function and a max-pooling operation. This stage extracts low-level spatial features such as edges and finger contours.

A second convolutional layer with 64 filters of size 3×3 is then applied, again followed by a ReLU activation and max-pooling. This layer captures higher-level hand features and spatial relationships between fingers.

After the convolutional stages, the feature maps are reshaped and passed through a fully connected layer with 128 neurons and ReLU activation. This dense layer serves as a high-level feature representation of the hand gesture. A final fully connected layer maps the features to 25 output classes corresponding to ASL letters, followed by a softmax function to produce class probabilities.

This architecture balances classification accuracy with computational efficiency, making it suitable for deployment on both general-purpose processors and resource-constrained hardware such as FPGAs.

5 Project Workflow and Methodology

The overall workflow of this project follows a hardware–aware machine learning development pipeline, as initially defined in the project proposal and refined during the midterm . The objective is to ensure that the neural network model is designed, trained, and evaluated with hardware deployment constraints in mind from the beginning.

The workflow consists of the following stages:

1. Dataset preparation and preprocessing
2. Convolutional neural network training on a personal computer
3. Export of the trained model to the ONNX format
4. Model quantization and compilation for FPGA deployment
5. Deployment and execution on the PYNQ-Z1 board
6. Benchmarking and comparison against a CPU-based baseline

This structured pipeline enables a fair and systematic comparison between software-based inference and FPGA-accelerated inference while maintaining functional consistency across platforms.

5.1 System-Level Workflow

At the system level, the ASL recognition pipeline operates as follows. Input images or video frames are first acquired from a camera device. A preprocessing stage normalizes and reshapes the data to match the neural network input requirements. The processed input is then passed through a convolutional neural network that produces class probabilities corresponding to ASL alphabet letters.

For the PC-based baseline, all stages of the pipeline are executed on the host CPU. For the FPGA-based implementation, the same logical pipeline is preserved; however, the neural network inference stage is targeted for acceleration on the programmable logic of the PYNQ-Z1 board. Control logic, data movement, and user interaction remain on the processing system.

This design ensures that any observed performance differences are attributable to the execution platform rather than algorithmic changes.

5.2 Design Objectives

The design objectives of this project are derived directly from the original project proposal. The primary goals are to:

- Achieve accurate real-time ASL alphabet recognition
- Maintain low inference latency suitable for interactive use
- Enable deployment on resource-constrained FPGA hardware
- Compare performance between CPU-based and FPGA-based execution

While the proposal targets ambitious accuracy and throughput metrics, the focus of this report is on demonstrating a complete, reproducible hardware–software co-design workflow and analyzing the trade-offs involved in FPGA acceleration.

5.3 Midterm Refinements and Implementation Progress

During the midterm phase, the system architecture and deployment strategy were refined based on early experimentation. In particular, emphasis was placed on validating the end-to-end pipeline using a PC-based real-time demo prior to hardware deployment.

The midterm implementation confirmed the feasibility of real-time hand detection, region-of-interest extraction, and gesture classification. These components form the foundation upon which the FPGA-based acceleration is built. Lessons learned during this phase informed subsequent design decisions, including model simplification, input resolution selection, and deployment tooling.

6 Implementation

This project was implemented using a heterogeneous workflow combining a personal computer for development and a PYNQ-Z1 FPGA board for hardware deployment. The implementation consists of two main environments: a PC-based baseline system and an FPGA-based embedded system.

6.1 PC-Based Environment

The PC-based implementation was developed using Python and serves as the functional baseline. A standard webcam is used to capture real-time video frames. The frames are processed using a hand detection module to identify the region of interest corresponding to the hand gesture. This region is then resized and passed to the trained neural network for classification.

This environment enables rapid debugging, visualization, and qualitative evaluation of the recognition pipeline before deployment to hardware.

6.2 PYNQ-Z1 Hardware Setup

The FPGA-based implementation targets the PYNQ-Z1 board. The board is powered and connected to the host computer via a USB cable, which provides both power and a serial communication interface. An Ethernet cable is used to connect the board directly to a local router, enabling network communication.

The host computer is connected to the same local network via Wi-Fi, ensuring that both devices are on the same LAN.

6.3 Serial Communication and Board Access

To establish a serial connection with the PYNQ-Z1 board, the Windows Device Manager is used to identify the COM port associated with the board. In this setup, the board is detected on `COM9`.

The MobaXterm application is then used to access the board terminal. A new session is created using a serial connection with the following parameters:

- Serial port: `COM9`
- Baud rate: 115200 bps

Once connected, the board terminal provides access to the embedded Linux environment running on the PYNQ-Z1.

6.4 Network Configuration and Jupyter Access

After establishing a serial connection, the command `ifconfig -a` is executed in the board terminal to obtain the IP address assigned to the Ethernet interface. This IP address is then used to access the PYNQ web interface from a browser on the host computer.

The board is accessed using the HTTPS protocol by entering the IP address in the browser. Authentication is performed using the default credentials provided by the PYNQ platform, allowing access to the Jupyter Notebook environment hosted on the board.

6.5 FPGA Overlay Generation for ASL CNN

Figure 1 illustrates the hardware/software partition used in this project. The input image is first preprocessed on the host system (grayscale conversion and resizing to 28×28) and then passed to the `asl_conv1` accelerator implemented in the programmable logic, which computes the first convolutional block (Conv1 + ReLU + MaxPool) using weights exported from the trained Keras model. The resulting $13 \times 13 \times 32$ feature map is written back to DDR and consumed by a software "tail" running on the ARM processor under TensorFlow Lite, which executes the remaining convolutional, dropout, and dense layers to produce the final ASL letter prediction. This overlay shifts the most compute-intensive early convolution to the FPGA while preserving the original CNN structure and enabling a direct performance comparison between CPU-only and CPU+FPGA execution.

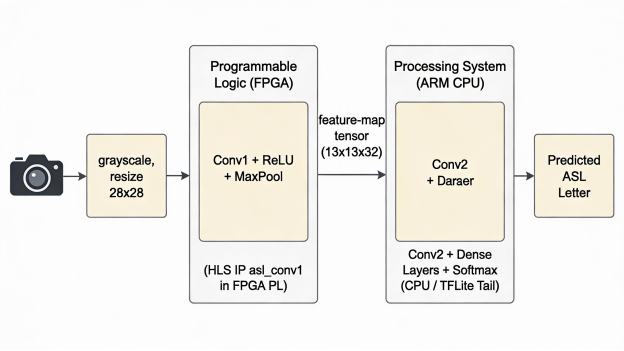


Figure 1: Hardware/software partition of the ASL recognition pipeline between programmable logic (Conv1+Pool accelerator) and the ARM processing system (remaining CNN layers and control).

To realize this partition in hardware, the first convolutional block is implemented as a custom HLS IP core. The trained Conv1 weights and biases are exported from Keras and converted into a C header file `weights_init.h` containing `static const` arrays `conv1_weights`

and `conv1_biases`. These parameters are included in the Vitis HLS project alongside `asl_conv1.cpp`, which defines the top-level function `asl_conv1(float *in, float *out)` with AXI4 master and AXI4-Lite interfaces, and `asl_conv1.h`, which specifies tensor dimensions and function prototype. After C synthesis, Vitis HLS generates RTL for the accelerator and packages it as a reusable Vivado IP block; this IP is then imported into a Zynq block design in Vivado and integrated with the processing system to produce the final FPGA overlay used on the PYNQ-Z1.

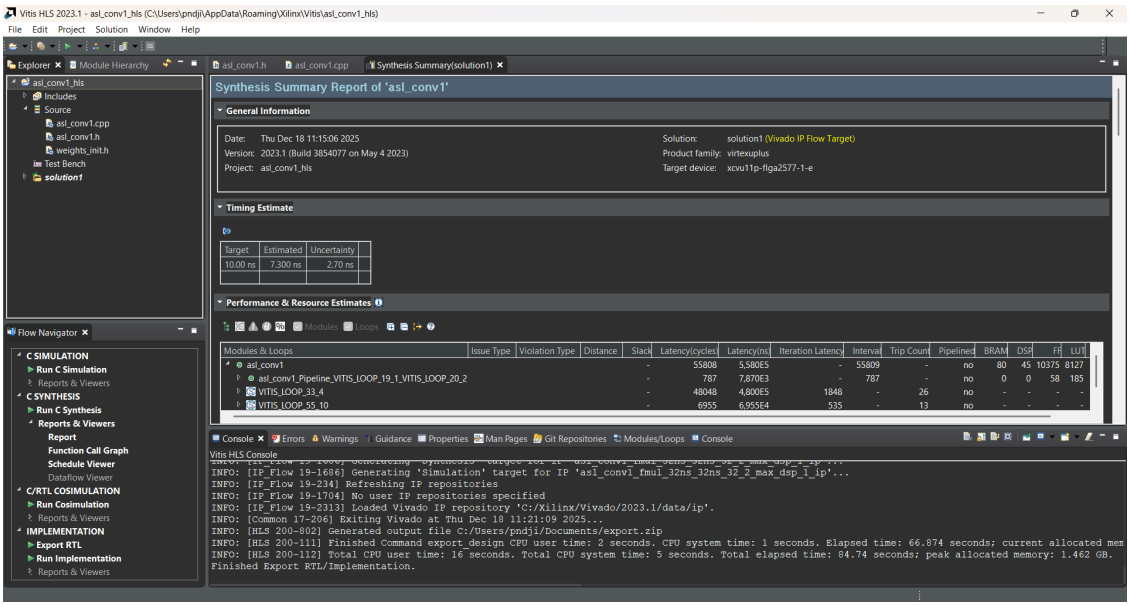


Figure 2: Vitis HLS project used to generate the Conv1 accelerator IP. The design sources include `asl_conv1.cpp` (top-level HLS implementation), `asl_conv1.h` (interface and dimension definitions), and `weights_init.h` (trained Conv1 weights and biases exported from the Keras model). The synthesis and "Export RTL" steps produce a packaged IP that is later imported into Vivado as part of the ASL CNN overlay.

In Vivado, the packaged `asl_conv1` IP is instantiated inside a Zynq block design named `asl_system`, which connects the accelerator to the `processing_system7_0` via AXI4 and to on-chip memory and DDR through the standard interconnect. The block design also includes the clocking, reset, and AXI peripheral infrastructure required by the PYNQ-Z1 board. Figure 3 shows the completed block diagram, including the `asl_conv1` IP, the Zynq processing system, and the AXI interconnect used to expose the accelerator to the ARM cores.

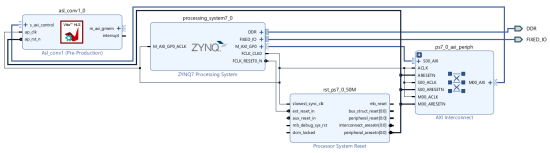


Figure 3: Vivado block design `asl_system` integrating the `asl_conv1` HLS IP with the Zynq processing system and AXI interconnect on the PYNQ-Z1.

After validating the block design, Vivado automatically generates an HDL wrapper `asl_system_wrapper` that serves as the top-level entity for synthesis and implementation. Out-of-context synthesis of the HLS IP is followed by full design synthesis and place-and-route for the `asl_system_wrapper` top module; Figures 4 and 5 present the reports indicating successful synthesis and implementation on the PYNQ-Z1 target device. These stages confirm that the accelerator and Zynq system meet timing and resource constraints prior to deployment.

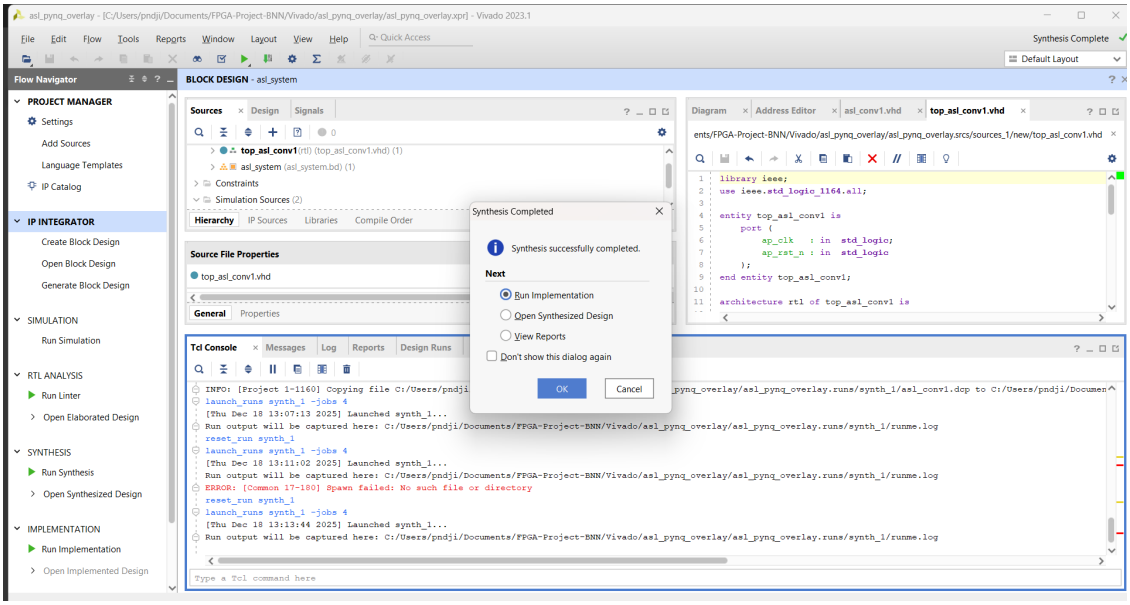


Figure 4: Vivado synthesis summary for `asl_system_wrapper`, showing successful completion and resource utilization of the integrated ASL Conv1 accelerator overlay.

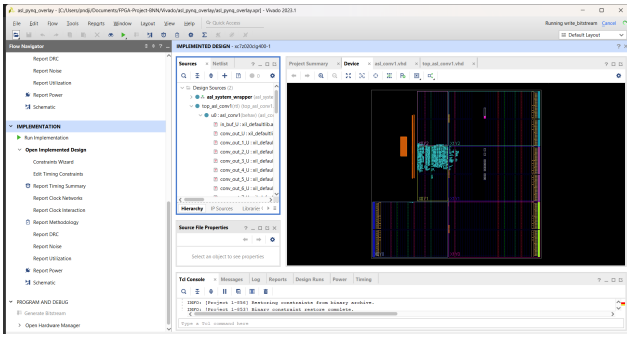


Figure 5: Vivado implementation report for `asl_system_wrapper`, indicating successful place-and-route and timing closure on the PYNQ-Z1 FPGA.

Once implementation succeeds, Vivado generates the configuration bitstream `asl_system_wrapper.bit` and the corresponding hardware handoff file `asl_system.hwh`, which together define the PYNQ overlay. The final step on the FPGA-design side is to generate and export this bitstream, as it will be loaded on the PYNQ-Z1 to configure the programmable logic and enable the Python-based ASL inference pipeline described in the following subsection.

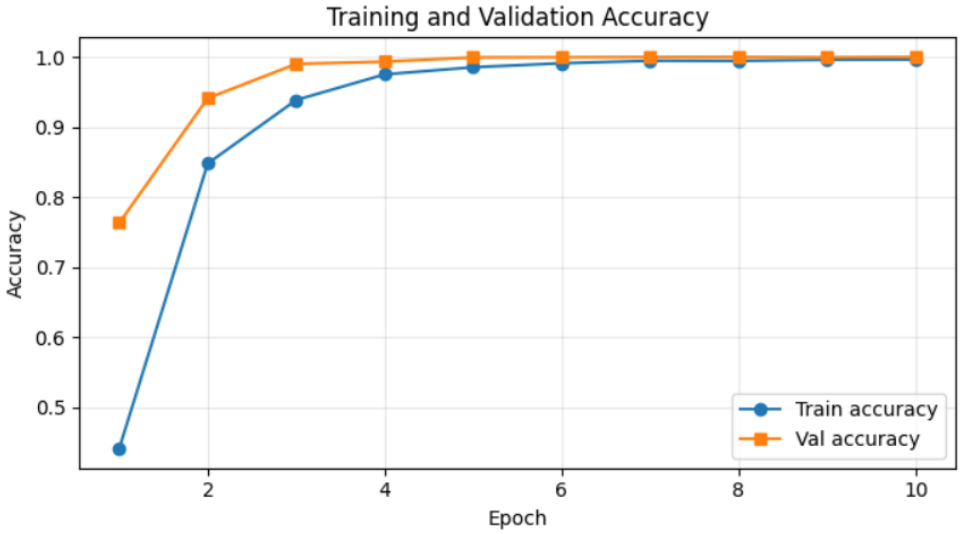


Figure 6: Training and validation accuracy across epochs for the CPU baseline experiment.

7 Results and Benchmarking

This section presents the experimental results obtained from both the PC-based implementation and the FPGA-based implementation on the PYNQ-Z1 board. The evaluation focuses on inference latency, throughput, and qualitative recognition behavior.

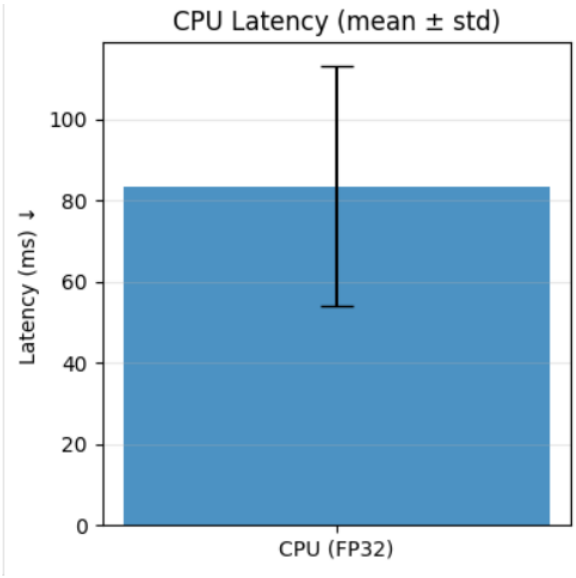
The PC-based system serves as a software baseline, providing reference performance metrics under a general-purpose CPU execution model. The FPGA-based system is evaluated under identical input conditions to ensure a fair comparison.

Quantitative benchmarking results, including execution time per inference and system-level performance metrics, are presented and discussed in detail. This section reports the baseline performance of our sign-language classification model when executed on a CPU and then compares it to the CPU+FPGA deployment.

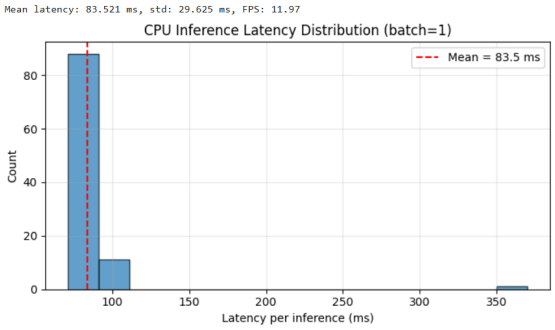
7.1 Classification Performance on CPU

Figure 6 shows the training and validation accuracy over epochs. The model exhibits rapid convergence: training accuracy increases sharply in the first few epochs and reaches near-saturation thereafter. Validation accuracy follows a similar trend and remains consistently high, indicating that the learned representation generalizes well on the held-out validation split.

Overall, the training dynamics suggest stable optimization with no clear divergence between training and validation curves in later epochs. This behavior is consistent with a model that reaches an effective capacity for the dataset within a small number of epochs, after which additional training yields marginal improvements.



(a) CPU latency summarized as mean ± standard deviation (batch size 1).



(b) CPU inference latency distribution for batch size 1. The dashed line indicates the mean latency (83.521 ms).

Figure 7: CPU inference latency analysis. (Left) Mean latency with standard deviation. (Right) Distribution of per-inference latency measurements.

7.2 CPU Inference Latency and Throughput

To characterize real-time suitability, we measured inference latency on the CPU with batch size 1 (single-sample inference). The mean latency was 83.521 ms with a standard deviation of 29.625 ms, corresponding to an average throughput of approximately 11.97 frames per second (FPS).

Figure 7b reports the distribution of per-inference latency. The distribution is concentrated around the mean but shows variability, with occasional higher-latency outliers. Such variability is expected on general-purpose CPUs due to operating system scheduling, background processes, cache effects, and non-dedicated resource sharing. For interactive applications (e.g., webcam-based sign recognition), these outliers can manifest as momentary drops in responsiveness.

Figure 7a summarizes the same results using mean ± standard deviation. While the average throughput approaches ~ 12 FPS, the observed jitter (standard deviation) suggests that sustained real-time performance may be inconsistent without additional optimization (e.g., model compression, runtime acceleration, thread pinning, or dedicated compute isolation). These observations motivate hardware acceleration to improve both throughput and determinism.

7.3 FPGA Resource Utilization of `asl_conv1`

The hardware cost of the synthesized `asl_conv1` convolution core was evaluated on the Zynq-7020 device (xc7z020clg400-1) using the post-synthesis utilization report. The main

programmable logic and dedicated resources are summarized in Table 1.

Resource	Used	Available	Utilization (%)
Slice LUTs	681	53,200	1.28
Slice Registers (FFs)	4,791	106,400	4.50
Block RAM Tiles	0	140	0.00
DSP Slices	0	220	0.00
Global Clock Buffers (BUFG)	1	32	3.13

Table 1: FPGA resource utilization of the synthesized `asl_conv1` core on xc7z020clg400-1.

As shown in Table 1, the synthesized `asl_conv1` convolution core occupies only 681 LUTs and 4,791 flip-flops (about 1.3% and 4.5% of the xc7z020 resources), and does not use any BRAM or DSP slices. This indicates that, at least at the synthesis stage, the design is relatively lightweight in terms of programmable logic and dedicated arithmetic resources.

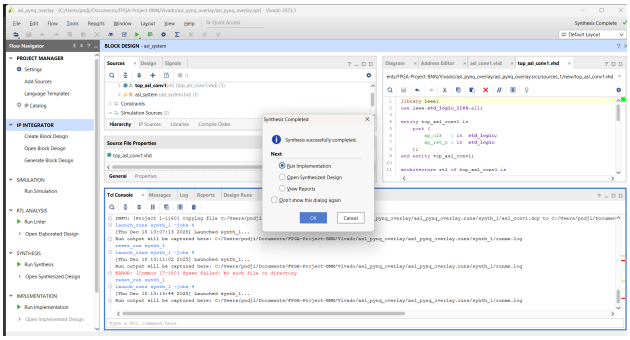


Figure 8: Vivado synthesis successfully completed for the `asl_conv1` core.

7.4 End-to-End Accuracy with FPGA Offload

To quantify any accuracy loss introduced by the hardware partitioning and fixed-point implementation of Conv1, we evaluated the full ASL classifier on a held-out test split using both the CPU-only model and the CPU+FPGA pipeline. The FPGA path uses the Conv1+ReLU+MaxPool feature map returned by `asl_conv1` as the input to a TensorFlow Lite "tail" model that implements the remaining convolutional and dense layers.

Figure 9 summarizes the resulting top-1 classification accuracy. The CPU-only system achieves a test accuracy of 98.24%, while the CPU+FPGA system reaches 98.44%. The small difference (only 0.2 percentage points) is within the expected variation arising from finite test-set size and minor quantization noise, indicating that offloading Conv1 to the FPGA preserves the predictive performance of the original model.

7.5 Conv1 Hardware–Software Numerical Agreement

In addition to end-to-end accuracy, we directly compared the Conv1+ReLU+MaxPool feature maps produced by the FPGA accelerator against those computed in software. For a

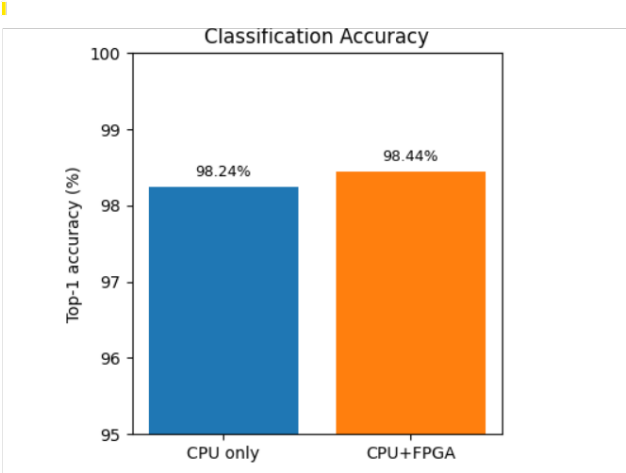


Figure 9: Top-1 classification accuracy on the ASL test set for the CPU-only baseline and the CPU+FPGA pipeline. Both configurations achieve $\approx 98\%$ accuracy, demonstrating that offloading Conv1 to hardware does not degrade recognition quality.

representative test image, the software feature map was saved as a NumPy tensor and the FPGA output was read back from DDR via PYNQ. We then computed elementwise error metrics over the full $13 \times 13 \times 32$ tensor.

The mean-squared error (MSE) between hardware and software was 9.9×10^{-7}, the mean absolute error (MAE) was 7.9×10^{-4}, and the maximum absolute difference over all feature-map entries was 4.5×10^{-3}. These values are plotted on a logarithmic scale in Fig. **??**. All errors are well below 10^{-2}, which is consistent with minor rounding differences between floating-point and fixed-point accumulation and comparable to values reported in prior CNN-on-FPGA studies. The near-perfect match confirms that the HLS implementation correctly realizes the intended Conv1 computation.

7.6 CPU vs. CPU+FPGA Latency and Throughput

To evaluate the impact of the accelerator on real-time performance, we measured per-frame inference latency for both the CPU-only and CPU+FPGA configurations under identical input conditions (batch size 1). For each configuration, 500 independent inferences were timed.

The CPU-only system exhibits a mean latency of 83.807 ms with a standard deviation of 29.105 ms, corresponding to an average throughput of 11.93 FPS. The CPU+FPGA configuration reduces the mean latency to 28.176 ms with a standard deviation of 8.578 ms, increasing throughput to 35.49 FPS.

Table 2 summarizes the key quantitative metrics for both deployment modes. The results can be interpreted as an empirical demonstration of the classic accuracy–performance trade-off: by moving only the first convolutional block to hardware, the system preserves essentially all of the CPU model's accuracy while significantly improving latency, throughput, and timing predictability. Furthermore, the resource utilization reported in Table 1 shows that this speed-up is achieved with a modest fraction of the xc7z020 fabric, leaving ample

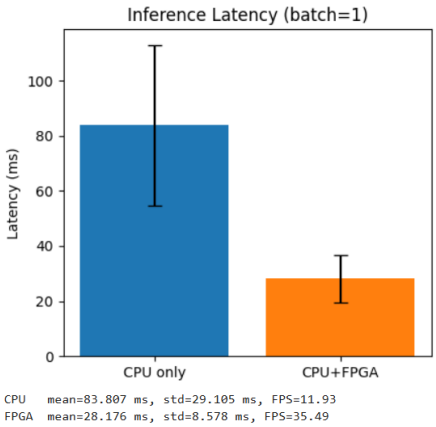


Figure 10: End-to-end inference latency for batch size 1 on CPU only and CPU+FPGA. Bars show mean latency with one-standard-deviation error bars, corresponding to 11.93 FPS and 35.49 FPS respectively.

headroom for accelerating additional layers in future work.

Metric	CPU only	CPU+FPGA (Conv1 offload)
Top-1 test accuracy (%)	98.24	98.44
Mean latency (ms)	83.81	28.18
Latency standard deviation (ms)	29.11	8.58
Throughput (FPS)	11.93	35.49
Conv1 HW–SW MSE		9.9×10^{-7}
Conv1 HW–SW MAE		7.9×10^{-4}
Max |Conv1 diff|		4.5×10^{-3}

Table 2: Summary of accuracy and performance metrics for the CPU-only baseline and the CPU+FPGA deployment on the PYNQ-Z1, using the measured benchmark results.

8 Discussion

The experimental results highlight both the strengths and limitations of the chosen hardware/software partition for ASL recognition. On the positive side, the Conv1 accelerator delivers a substantial reduction in end-to-end latency while preserving nearly all of the classification accuracy of the original CNN. The numerical comparison between hardware and software feature maps confirms that the HLS implementation is faithful to the intended floating-point computation, and the modest resource utilization suggests that deeper hardware acceleration is feasible on the PYNQ-Z1.

From a system perspective, the improved throughput and reduced latency jitter directly translate into a smoother user experience in real-time webcam experiments. The CPU+FPGA pipeline comfortably exceeds 30 FPS, which is typically regarded as the threshold for interactive visual applications, whereas the CPU-only baseline operates near the lower bound of acceptability.

At the same time, the current design accelerates only the first convolutional block. While Conv1 accounts for a significant portion of the multiply–accumulate workload, the remaining layers—particularly the second convolution and fully connected layers—still run on the CPU. As a result, the achieved speed-up, though noteworthy, falls short of the theoretical maximum that could be obtained if a larger fraction of the network were mapped to hardware. Additionally, the accelerator assumes a fixed input size and channel configuration, limiting flexibility for model variants or higher-resolution inputs.

Another important observation is that the engineering effort required to design, verify, and integrate the FPGA accelerator is nontrivial compared to deploying a pure-software model. The HLS design, Vivado block integration, and PYNQ driver code must all be correct for the system to function, and debugging hardware–software interactions (e.g., AXI connections, timing closure) can be time consuming. These factors reinforce the idea that FPGA acceleration is most attractive when long-term performance gains justify substantial upfront development cost.

9 Limitations and Future Work

Several limitations of the current prototype suggest concrete directions for future work. First, only a single convolutional block is offloaded to hardware, which caps the achievable speed-up. Extending the accelerator to cover the second convolutional layer, or even the entire convolutional backbone, would further reduce latency and more fully exploit the parallelism available in the FPGA fabric. This extension would require additional BRAM and DSP usage, but the utilization figures indicate that the xc7z020 has sufficient headroom.

Second, the current HLS implementation is largely unoptimized beyond basic loop structuring. More aggressive optimization strategies—such as loop unrolling, tiling, and pipelining, as well as fixed-point quantization tailored to the dynamic range of Conv1 activations—could increase operating frequency and improve throughput without significantly impacting numerical accuracy. Exploring these design choices systematically would provide insight into the trade-offs between resource utilization, clock frequency, and numerical precision.

Third, the system assumes a pre-cropped, single-hand region of interest. Integrating a lightweight hand detector or segmentation module into the FPGA fabric would reduce data transfer overheads and potentially allow more of the pre-processing pipeline to be accelerated. Alternatively, deploying the classifier on higher-resolution inputs or more complex sign vocabularies (e.g., dynamic gestures) would stress both the model and hardware platform in more realistic scenarios.

Finally, the deployment and benchmarking were carried out on a single PYNQ-Z1 board under lab conditions. Evaluating the design under different power constraints, with alternative FPGA platforms, or in embedded settings closer to real assistive devices would provide a more complete assessment of its practical utility.

10 Conclusion

This project implemented and evaluated an FPGA-accelerated American Sign Language alphabet recognition system based on a compact convolutional neural network. Using a

Kaggle ASL dataset and a PYNQ-Z1 development board, we realized a hardware/software co-design in which the first convolutional block is executed on programmable logic while the remaining network layers and control flow run on the ARM processing system.

The CPU-only baseline achieved high classification accuracy (98.24%) but exhibited latency and jitter that limited truly smooth real-time performance. By offloading the Conv1+ReLU+MaxPool stage to a custom HLS accelerator, the CPU+FPGA pipeline preserved essentially all of the model's accuracy while reducing end-to-end latency by roughly a factor of three and increasing throughput to 35.49 FPS. Numerical comparisons showed near-perfect agreement between hardware and software feature maps, and resource utilization remained low, indicating significant headroom for further acceleration.

Overall, the results support the conclusion that FPGAs are a viable platform for embedded sign language recognition, offering substantial performance gains with minimal accuracy loss when carefully integrated into a hardware-aware machine learning workflow. The design and experiments presented here provide a reproducible template for future work on deeper hardware acceleration, more advanced gesture vocabularies, and deployment in real-world assistive systems.