A Compiler-Aware Framework for Partitioned Neural Network Inference on FPGA DPUs

Federico Buccellato, Luca Mannini, Corrado De Sio
Politecnico di Torino
Turin, Italy
federico.buccellato@polito.it, s291065@studenti.polito.it, corrado.desio@polito.it

***Abstract*—The increasing adoption of Artificial Intelligence is driving the development of larger and more accurate neural networks. However, their high computational cost leads to significant inference latency, especially on resource-constrained embedded platforms such as FPGA-based systems. To address this issue, AMD introduced the Deep Learning Processing Unit, an accelerator integrated into the Vitis AI toolchain for efficient execution of quantized neural networks. Nevertheless, models with many parameters can be difficult to deploy on a single board due to latency or capacity constraints. In these scenarios, splitting a model into separately compilable fragments becomes useful for more flexible deployment. This process is not immediate within the Vitis AI flow. Our analysis shows that manually partitioning a network into independently compiled sub-networks can remove compiler optimizations. In particular, losing the global view of the quantized XIR graph can prevent the toolchain from preserving DPU mapping, moving accelerable operations to the CPU and causing severe performance degradation. Based on this observation, we analyze the Vitis AI compiler and propose an XIR-level splitting framework that generates independently compilable `.xmodel` fragments while preserving the context required for DPU mapping. The approach keeps the workflow high-level, without requiring advanced hardware design expertise or low-level design changes. Experimental results on CNN and ConvViT-based models show that naive splitting can introduce slowdowns up to ×249 on CNNs and ×542 on ConvViT variants. The proposed framework restores correct DPU mapping by addressing boundary-context loss and incomplete dependency collection, bringing latency back to the expected range for hardware-accelerated execution.**

***Index Terms*—FPGA, Distributed Inference, Deep Learning, DPU, Vitis AI, XIR**

I. Introduction

The growing adoption of Artificial Intelligence is driving the development of increasingly accurate and complex models, often based on deep neural networks with a large number of parameters [1]. Although these models achieve very high performance in many application domains, their computational cost represents a concrete limitation when they are deployed on embedded platforms. In these scenarios, strict latency constraints and limited available resources make the efficient execution of large models difficult.

FPGA-based architectures have emerged as a promising solution for edge inference, due to their flexibility, reconfigurability, and good trade-off between performance and energy efficiency. To exploit these features for neural network execution, FPGA-specific accelerators have been developed to optimize the most common operations in deep learning models, such as convolutions, matrix multiplications, and activation functions. In this direction, solutions such as the Deep Learning Processing Unit (DPU), introduced by AMD within the Vitis AI toolchain, represent an example of integration between hardware acceleration and high-level development flows, enabling a more accessible deployment compared to fully custom approaches.

Despite these advances, the continuous increase in model complexity often makes the use of a single FPGA board insufficient. Models with a large number of parameters may exceed the available resources or fail to meet the required latency constraints, making it necessary to distribute the execution across multiple devices. This requirement introduces new challenges related to model partitioning and the efficient management of distributed execution.

Traditionally, the problem of model parallelism and distribution has been widely studied in the context of high-performance platforms, especially GPUs and computing clusters. In these environments, techniques such as model parallelism and pipeline parallelism have been extensively explored and optimized, supported by mature software frameworks and dedicated infrastructures [2]. In contrast, in edge scenarios and on FPGA architectures, this problem is less explored and is often addressed through heterogeneous and poorly standardized solutions.

Existing approaches for distributed deployment on FPGAs are frequently based on custom implementations, which require manual design of inter-device connections, explicit management of data transfers, and deep knowledge of the underlying hardware [3]. This makes such solutions difficult to scale and less accessible in real application scenarios.

In this context, although accelerators such as the DPU have simplified FPGA deployment in single-device configurations, extending this approach to partitioned models remains problematic. In this work, we observe that a direct approach based on manually splitting the model into independently compiled sub-networks can lead to a loss of optimizations, since the compiler no longer has a global view of the network. This behavior can severely degrade performance, with slowdowns reaching up to ×542 compared to the optimized execution.

Starting from this analysis, we systematically study the behavior of the Vitis AI toolchain in model partitioning scenarios, showing that naive sub-network compilation can break the global context required for correct DPU mapping. To address

Manuscript received May 25, 2026; revised July 15, 2026; accepted July 13, 2026. Published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.144

this issue, we propose an XIR-level splitting framework that partitions models at user-defined points while preserving the optimizations of the original compilation flow. The framework remains integrated in the high-level Vitis AI pipeline and produces fragments that can be coordinated through standard mechanisms such as TCP or SLURM. Experimental results show that the proposed approach recovers the optimizations lost by naive splitting and restores latency to the expected range for DPU-accelerated execution.

II. Related Works

With the increasing adoption of Artificial Intelligence, neural networks have reached larger sizes and higher complexity, increasing both memory requirements and the computational cost of inference. This growth has made it necessary to develop solutions able to support deeper and heavier models without compromising latency, throughput, and energy efficiency.

One of the most widely studied directions concerns the distribution of the computational workload across multiple resources. In high-performance systems, this problem has mainly been addressed on GPUs and computing clusters. Techniques such as data parallelism, model parallelism, and pipeline parallelism have been used to scale the execution of large neural networks by exploiting infrastructures with high computational capacity and high-bandwidth interconnections. Works such as DistBelief showed the possibility of executing large-scale models in distributed environments [2], while later approaches such as GPipe and PipeDream investigated pipeline parallelism for splitting deep networks into multiple execution stages [4], [5]. These solutions have played a central role in the scalability of neural models, but they are mainly designed for datacenter or HPC environments, which are very different from the embedded and edge scenarios considered in this work.

In the edge context, the problem has different characteristics. Embedded platforms must satisfy stricter constraints in terms of latency, energy consumption, available memory, and hardware cost. In this scenario, FPGAs are a particularly interesting platform for inference, due to their reconfigurability and their ability to implement specialized architectures for specific neural workloads. Several works have shown that FPGA accelerators can achieve good trade-offs between performance and energy efficiency for convolutional neural networks and quantized models [6]. However, obtaining these benefits often requires careful design of the architecture, memory management, and operation mapping on the device [7], [8].

To address the growth of modern models and the limited resources available on a single device, parallel approaches on edge and multi-FPGA platforms have also been studied. Some works have explored FPGA clusters for specific applications, such as recommendation system inference, using distributed pipelines and dedicated interconnections [3], [9]. Other multi-FPGA approaches, including OpenCL-based solutions, try to raise the abstraction level compared to traditional hardware design, but still require explicit management of partitioning, communication, and execution across devices. Overall, distributed execution on FPGAs often remains tied to custom solutions, which are difficult to generalize and not straightforward to integrate into a standard deployment pipeline.

Given the complexity of FPGA development and the limited generality of many existing custom solutions, several frameworks have been proposed in recent years to automate the generation of neural accelerators. In this context, FINN introduces a flow for quantized and binarized networks based on specialized dataflow architectures [10]. DNNWeaver generates synthesizable accelerators from high-level network descriptions [11], while fpgaConvNet addresses the automatic mapping of convolutional networks onto embedded FPGAs by optimizing the computational graph according to the available resources [12]. These toolflows highlight the importance of automation, but in many cases they still produce custom architectures or require expertise in hardware synthesis, accelerator configuration, and resource optimization.

To make FPGA acceleration more accessible also to users without specific hardware design expertise, the AMD ecosystem provides the Deep Learning Processing Unit integrated into the Vitis AI toolchain [13]–[16]. Compared to manually designed accelerators, the DPU allows a higher-level workflow, in which the model is quantized, compiled, and executed using tools already integrated in the toolchain. In this way, acceleration of quantized neural networks can be used without manually designing the accelerator datapath or directly modifying the hardware design.

However, the problem of applying this paradigm to modern large-scale models remains open, especially for networks with an increasing number of parameters, more complex structures, and latency constraints that are difficult to satisfy on a single board.

In these cases, a natural solution is to partition the neural network into multiple sub-models, so that each portion can again be managed within the Vitis AI flow for DPU execution. However, as shown in this work, manually splitting the network into independently compiled sub-networks can break the context used by the compiler and lead to the loss of the optimized mapping on the accelerator. This creates an intermediate issue between deployment and compilation. Splitting the model is not sufficient by itself; the split must preserve the graph information that allows the toolchain to maintain the original optimizations.

This work addresses this issue. Starting from a study of the Vitis AI toolchain behavior on partitioned models, we propose an XIR-level splitting framework that allows a network to be divided at user-defined points and generates independently compilable `.xmodel` fragments. Unlike approaches based on custom accelerators or specialized distributed runtimes, the framework keeps Vitis AI and the DPU as the starting point and operates on the intermediate graph to preserve the optimizations of the original compilation.

The framework reconstructs the quantization context at fragment boundaries and verifies the structural completeness of the extracted graph, including the dependencies required by branches, residual connections, and aggregation points. The evaluation was conducted on two model families, Parallel-

ExpertsNet and ConvViT, showing that naive splitting can cause slowdowns up to ×249 on CNN models and up to about ×542 on ConvViT models. In both cases, the framework restores effective DPU mapping and brings latency back to the expected range for hardware-accelerated execution. In this way, model partitioning remains compatible with a high-level workflow, without modifying the DPU or making low-level changes to the hardware design, and the generated fragments can be coordinated through standard mechanisms such as TCP or orchestration systems such as SLURM.

III. Technical Background

This section introduces the main tools and technological components used in this work. Since the proposed framework operates within the Vitis AI compilation and deployment flow, we describe the Deep Learning Processing Unit, the Vitis AI toolchain, the Xilinx Intermediate Representation, and SLURM as a workload manager for cluster environments.

A. Deep Learning Processing Unit

The AMD Deep Learning Processing Unit (DPU) is a dedicated hardware accelerator provided as a preconfigured overlay for FPGA and SoC platforms [13]. It is designed to accelerate neural network inference on AMD reconfigurable systems, especially for quantized models. By exploiting the parallelism of the FPGA fabric, the DPU reduces CPU involvement during the most computationally intensive stages of inference.

The DPU uses a parallel execution model based on specialized pipelines for common neural network operations, including convolutions, pooling, activation functions, and matrix multiplications. An internal control engine manages on-chip dataflow and coordinates computation through an optimized memory hierarchy. Intermediate results can be stored locally, while weights and larger data structures are placed in external DDR memory. This organization enables high throughput and low inference latency while preserving a higher abstraction level than fully custom accelerators.

Although the DPU is configurable for different performance and resource constraints, its internal implementation is not normally exposed at RTL level. It is provided as a preconfigured component within the Vitis AI flow, so users mainly interact with model preparation, quantization, compilation, and deployment. This simplifies integration, but limits direct changes to the hardware microarchitecture.

B. Vitis AI Toolchain

Vitis AI is the AMD toolchain for deploying neural networks on DPU accelerators [14]. The flow starts from a model trained with common frameworks, such as PyTorch or TensorFlow, and transforms it into an executable representation for the target FPGA platform. Its main components are the quantizer and the compiler.

The quantizer converts floating-point models into reduced-precision formats, typically integer representations, suitable for efficient DPU execution. During this step, the model is calibrated with a representative dataset to estimate scales, numerical ranges, and quantization parameters for weights and activations. This phase affects both final accuracy and graph consistency.

The Vitis AI compiler analyzes the quantized model and generates the executable graph for the target platform. As part of this process, it produces compiled `.xmodel` artifacts that are compatible with execution on the DPU. DPU-compatible operations are identified and grouped into accelerable subgraphs, while unsupported operations are assigned to the host CPU. In the standard flow, the compiler processes the complete network and can apply global optimizations for efficient accelerator mapping. Overall, Vitis AI links the trained model to hardware execution through quantization, calibration, compilation, and the generation of deployment artifacts for DPU execution.

C. Xilinx Intermediate Representation

The Xilinx Intermediate Representation (XIR) is the graph-based intermediate representation used in Vitis AI to describe neural networks targeting hardware execution [17]. Each node represents an operation and stores metadata such as quantization parameters, tensor attributes, and compilation-related information. This structure preserves a consistent model representation when moving from high-level frameworks to the FPGA platform.

During compilation, the XIR graph is analyzed to identify subgraphs executable on the DPU and portions that must remain on the host CPU. XIR also supports optimizations such as operator fusion, for example merging convolution, normalization, and activation sequences into computation units better suited to hardware execution. Its APIs allow programmatic inspection and manipulation of the graph, including topology navigation, access to node attributes, and modification of the intermediate representation.

D. SLURM Workload Manager

SLURM is an open-source workload manager for cluster environments [18]. It allocates computational resources, launches processes on available nodes, and coordinates parallel or distributed applications. In a cluster, SLURM allows users to request nodes, CPUs, memory, accelerators, or other resources, and to execute programs through commands such as `srun`.

Operationally, SLURM separates resource management from the executed application. The system selects nodes, prepares the execution environment, and starts the processes, while communication between them remains managed by the application or runtime framework. This separation is useful when experiments must be launched, supervised, and repeated in a controlled way without embedding cluster-management logic directly into the application code.

IV. Methodology

The proposed framework automates the generation of DPU-optimized `.xmodel` fragments starting from a neural network, a calibration dataset, and a set of user-defined cut points. The objective is to provide a pipeline that, given the cut points, performs quantization, calibration, splitting, boundary correction, graph verification, and fragment compilation, while preserving as much as possible the optimizations obtained on the complete model.

A. Empirical Characterization of Fragment Compilation

Before defining the splitting framework, an empirical characterization of the Vitis AI compiler was carried out to observe the behavior of the toolchain when quantized XIR graphs are divided into independent fragments. For this purpose, about thirty heterogeneous small-to-medium micro-models were created. They were designed to reproduce common computational patterns in neural networks and to highlight the effects of splitting on sequential, branched, and residual structures.

The micro-models were organized into two main groups. The first group included linear convolutional patterns, such as `Conv-BN-ReLU` sequences, `Conv-ReLU` blocks, pooling operators, and combinations of consecutive layers. The second group included non-linear structures, such as parallel branches, residual connections, convergence points, and aggregation operators. This selection made it possible to evaluate the compiler behavior both on simple flows, where dependencies follow an almost sequential path, and on graphs where operator relations are distributed across multiple paths.

For each micro-model, the complete graph was compiled first and used as a reference to measure the mapping of operations onto the DPU. The same model was then split at intermediate or critical points of the graph. The cut points were selected to cross regions that are relevant for the compiler, for example between convolution and activation, between batch normalization and activation, between pooling and the following convolution, inside a parallel branch, or near aggregation nodes such as `add` and `concat`. The resulting fragments were then compiled separately and compared with the compiled version of the complete model.

This procedure made it possible to directly analyze which XIR graph transformations preserve efficient DPU mapping and which ones introduce optimization loss. In the case of the complete model, the Vitis AI pipeline has access to the whole quantized XIR graph. The compiler can therefore apply global optimizations, such as compatible-operator fusion, consistent management of numerical conversions, and mapping of supported regions onto the DPU. In this configuration, the ARM CPU is generally limited to secondary or unsupported operations, while the main computational workload is executed on the accelerator.

During the characterization, `fix` nodes emerged as a key element for the correct interpretation of fragments. These nodes do not correspond to layers of the original model, such as convolutions or activations, but are XIR operators introduced during quantization. They mark the transition of tensors to the fixed-point domain used by the DPU and store numerical metadata, including the `fix_point` parameter required to correctly interpret quantized values. Therefore, the presence of a `fix` node indicates that a tensor belongs to the quantized flow compatible with the accelerator.

When the graph is split, some tensors that were internal in the complete model become inputs or outputs of the fragments. During this step, part of the numerical and structural information associated with these tensors may remain outside the extracted sub-network. The metadata expressed by `fix` nodes, or implicitly encoded in producer–consumer relations, may be located in the previous or following fragment. As a consequence, a region that is fully compatible with the DPU in the complete model may be interpreted as incomplete when compiled in isolation.

TABLE I
DPU UTILIZATION FOR REPRESENTATIVE COMPILER PATTERNS.

Pattern	Cut	Complete (%)	Split (%)
Conv-BN-ReLU	BN/ReLU	71.4	50.0
Conv-ReLU	Conv/ReLU	71.4	50.0
Pooling	Pool/Conv	75.0	73.3
Branch split	In branch	85.7	63.6
Branch join	Add/Concat	88.2	68.0

This behavior was evident in cuts applied inside sequential patterns recognized by the compiler. In micro-models based on `Conv-BN-ReLU` or `Conv-ReLU` blocks, splitting between tightly connected operators reduced the compiler's ability to reconstruct the same fusions obtained on the complete graph. In these cases, the fragment boundary interrupts a relation that, in the full network, contributes to the generation of a more compact DPU region. Separate compilation may therefore introduce additional conversions or assign to the CPU operations that, in the complete model, were included in the accelerated subgraph.

In micro-models with branches, residual connections, and convergence points, the optimization loss was instead associated with the removal of lateral dependencies. In these graphs, the information required to correctly interpret a fragment is not always located along the main path delimited by the cut points. Operators such as `add`, `mul`, `scale`, or `concat` may depend on tensors produced along parallel paths. If these nodes or their predecessors are excluded, the compiler receives a structurally incomplete fragment and may not recognize some accelerable regions.

As shown in Table I, fragment compilation produces, in several cases, a reduction in DPU utilization compared with the compilation of the complete model. In the `Conv-BN-ReLU` and `Conv-ReLU` patterns, cutting at an intermediate point of the block reduces the percentage of operations mapped onto the accelerator. In branched cases, the reduction is mainly related to the loss of structural dependencies required to correctly reconstruct the original graph. Therefore, the cut point does not act only as a topological separation of the graph, but also changes the context used by the compiler to apply its optimizations.

The analysis of the results shows that the loss of acceleration does not necessarily depend on the presence of operators unsupported by the DPU. In several micro-models, the same operations are accelerated when the complete graph is compiled, but are partially assigned to the ARM Cortex-A53 when the model is split and compiled as independent fragments. Since the ARM processor is not optimized for highly parallel quantized workloads, this fallback can introduce a significant latency bottleneck.

This behavior is caused by the different level of information available during compilation. With the complete graph, the

compiler has access to the global relations between operators, the quantization nodes, and the dependencies between parallel paths. With independent fragments, some of this information may be located beyond the fragment boundaries. Under these conditions, the toolchain adopts a conservative strategy and prioritizes compilation correctness over maximum acceleration.

The characterization performed on the micro-models guided the design choices of the proposed framework. The results highlighted the need to perform splitting after global quantization, to explicitly reconstruct the numerical context at fragment boundaries, and to complete each subgraph with the required dependencies before compilation. The framework was therefore developed to generate independent fragments that preserve the structural and quantization information available in the complete model.

B. Post-Quantization XIR Fragmentation

After analyzing the behavior of the Vitis AI compiler when the graph is partitioned, a splitting framework was developed to reduce the loss of optimizations during independent fragment compilation. The framework takes as input the original FP32 model, a calibration dataset representative of the target application, and a set of user-defined cut points. Starting from these inputs, it generates independent XIR fragments intended for DPU execution, while preserving the quantization context and structural information available during complete-graph compilation.

As shown in Fig. 1, the proposed framework follows the standard Vitis AI flow and extends it with an intermediate phase dedicated to controlled XIR graph splitting. The FP32 model is first quantized and calibrated using the calibration dataset. This step produces a quantized XIR graph in which the numerical information required for DPU execution is derived from the complete model. Only after this global quantization phase does the framework apply the selected cut points, extract the corresponding fragments, reconstruct their boundaries, and prepare them for the following verification and generation stages.

Performing the split after quantization is a central design choice. If the network were partitioned before calibration, each fragment would be quantized separately, losing the common numerical reference of the original model. By quantizing the whole network first, the framework can recover from the complete XIR graph the quantization parameters needed to build fragments that are consistent with the fixed-point representation used by the DPU.

Starting from the quantized XIR graph, the framework analyzes the producer–consumer relations between operators. For each portion delimited by the selected cut points, the computational nodes belonging to the corresponding execution path are identified. These nodes form the initial core of the fragment. The framework then adds the auxiliary elements required for the correct interpretation of the fragment, such as constants, parameters, and quantization-related operators.

The boundary reconstruction phase restores the quantization context that may be lost when internal tensors become

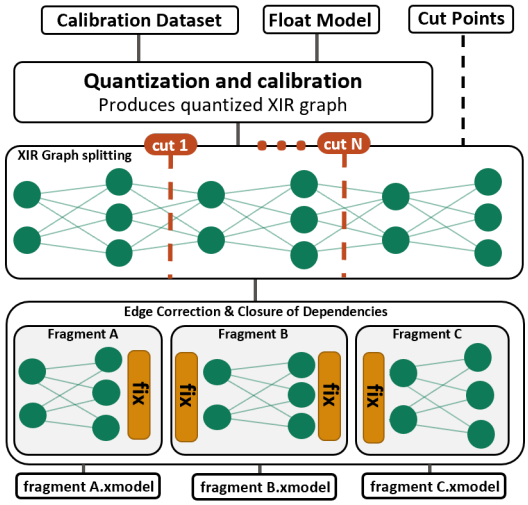

Fig. 1. Flow of the proposed framework for quantization, splitting, correction, and compilation of the fragments.

fragment inputs or outputs. Instead of treating these tensors as generic external values, the framework uses the quantization metadata available in the complete XIR graph to reconstruct the corresponding boundary representation. In practice, synthetic `fix` nodes are inserted at the input and output boundaries of each fragment, with parameters recovered from the original graph. For output boundaries, the framework traces the original computational path backward from the fragment output and uses the closest compatible `fix_point` value found in the complete graph. This allows each fragment to preserve a numerical representation consistent with its original position in the network.

After boundary reconstruction, the framework produces independent XIR fragments whose inputs and outputs are consistent with the quantization context of the original graph. Each fragment contains the selected computational core, the auxiliary nodes required for its correct interpretation, and the synthetic `fix` nodes inserted at the boundaries.

C. Fragment Completion and Structural Validation

After splitting and boundary reconstruction, the framework performs a structural verification of the generated fragments. This phase checks that each extracted graph is complete and can be serialized as an independent XIR model. In particular, it verifies that all operations required by the fragment are present in the generated graph and that no unresolved references remain toward portions excluded by the cut.

When a portion of the graph is extracted, it is not sufficient to copy only the operators located between two cut points. Each operation may depend on other nodes required for its execution, such as weights, biases, constants, quantization nodes, or results produced by previous operations. If one of these elements is not included in the fragment, the resulting graph may be incomplete or not correctly interpretable by the toolchain.

To avoid this issue, the framework performs dependency completion starting from the main set of operators assigned to the fragment. For each collected node, the incoming edges

are inspected in the original graph to identify any predecessors required to produce its inputs. If a required predecessor has not yet been copied, it is added to the fragment. The procedure is repeated until a complete iteration does not identify any new nodes to add to the fragment. Since the XIR computational graph is acyclic, this process terminates once all required dependencies have been included.

This phase is particularly useful in networks with residual connections, parallel branches, or local convergence points, where some dependencies may not lie on the main path of the fragment, while still being necessary for correct graph reconstruction.

Once the dependencies have been completed, the collected operations are ordered consistently with the graph dataflow. The framework therefore uses Kahn's algorithm to perform a topological ordering of the operators [19]. For this purpose, the in-degree of each node in the completed fragment is computed, and the ordering is initialized with the nodes that have no unresolved predecessors. These nodes are progressively removed from the working set, while the in-degree of their successors is updated. The resulting order defines the sequence in which operations are recreated in the new XIR graph, ensuring that each node is inserted only after the nodes it depends on.

This step ensures that the generated XIR graph is consistent, serializable, and correctly interpretable by the Vitis AI toolchain. Since the procedure is performed during offline fragment generation, it does not affect inference latency.

Once structural verification is complete, each fragment is saved as an independent XIR graph. The result of this stage is a collection of self-contained XIR models, each corresponding to a portion of the original network and ready for compilation through the Vitis AI toolchain. At the end of the process, the framework also generates the compiled models for DPU execution and creates a configuration file that describes the relation between the sub-models, specifying how the produced fragments must be connected and interpreted with respect to the structure of the original network.

V. Experimental Analysis

The results presented in this section validate the proposed framework from the perspective of DPU mapping. The analysis compares the behavior of the compiler with and without the corrections introduced by the framework, evaluating their impact on the distribution of operations between DPU and CPU and on the resulting execution latency. The evaluation considers two model families, a CNN with parallel branches and a ConvViT architecture, in order to assess the framework both on mainly convolutional partitions and on graphs with residual connections and more complex dependencies. The compatibility of the generated fragments with standard execution mechanisms, such as TCP and SLURM, is also evaluated.

A. Experimental Setup

The experiments were conducted on a cluster of three AMD Xilinx Kria KV260 boards, each equipped with a DPU DPUCZDX8G B4096 running at 300 MHz and a quad-core ARM Cortex-A53 processor at 1.33 GHz with 4 GB of DDR4 RAM. The boards are connected through a Gigabit Ethernet local network. The system software is Ubuntu 22.04 LTS (ARM64), with Vitis AI 3.5.0, VART and XIR in the same version, Python 3.10, and SLURM 21.08.5 for process orchestration. A separate x86 machine, external to the cluster, was used as a compilation server to run the Vitis AI toolchain inside a Docker container.

The first evaluated model family is ParallelExpertsNet, a pure CNN architecture with three parallel branches. All models accept input tensors with shape [1, 224, 224, 3] and produce an output with 10 classes. Four variants of increasing size were considered, whose architectural characteristics are reported in Table II.

TABLE II
Architectural characteristics of the evaluated CNN variants.

Variant	Input Size	Parameters (M)	Depth	Max Channels
Small	[1, 224, 224, 3]	∼1.3	12	128
Medium	[1, 224, 224, 3]	∼26	18	512
Large	[1, 224, 224, 3]	∼133	20	1024
Huge	[1, 224, 224, 3]	∼364.4	22	1536

The second evaluated family is based on ConvViT, with convolutional blocks and vision-transformer-like components. This architecture introduces a more complex graph than the pure CNN, including residual connections and aggregation operations that make the correct handling of internal fragment dependencies more relevant. The characteristics of the three considered variants are reported in Table III.

TABLE III
Architectural characteristics of the evaluated ConvViT variants.

Variant	Input Size	Parameters (M)	Embed Dim
Small	[1, 224, 224, 3]	∼2.3	192
Medium	[1, 224, 224, 3]	∼20	384
Large	[1, 224, 224, 3]	∼100	768

Each benchmark configuration was executed for 25 iterations. The first 5 iterations were used as warm-up and discarded from the final measurement, while the reported latencies correspond to the arithmetic mean of the following 20 executions. For each variant, fragments compiled without the framework, obtained through naive splitting, were compared against fragments generated with the proposed framework. The evaluation considers the percentage of operations mapped to the DPU and the overall execution latency. To verify the compatibility of the pipeline with standard orchestration tools, the same corrected fragments were executed both with manual launch and direct TCP communication, and through SLURM with FPGA resource allocation using the GRES mechanism.

B. Experimental Results

Before analyzing the effect of the framework on operation mapping, the compatibility of the pipeline with standard execution mechanisms was verified. This check was performed on the CNN models of the ParallelExpertsNet family, using the `.xmodel` fragments produced by the framework. The fragments were executed both with a manual TCP-based setup and through SLURM for process launch and orchestration. The goal of this comparison is to show that, once correctly compiled fragments are generated, their use does not require

dedicated hardware mechanisms or complex communication infrastructures.

TABLE IV
LATENCY COMPARISON BETWEEN MANUAL TCP AND SLURM-BASED ORCHESTRATION.

Variant	TCP (ms)	SLURM (ms)	Rel. (%)
Small	3.52	3.64	+3.4%
Medium	50.75	50.86	+0.2%
Large	122.65	122.54	-0.09%
Huge	322.89	323.02	+0.04%

The results in Table IV show that the two execution modes produce almost equivalent latencies. The relative differences are close to zero for Medium, Large, and Huge, while they are slightly more visible for Small, where the very low absolute latency makes any marginal variation more noticeable. This indicates that using an orchestrator such as SLURM is compatible with the proposed pipeline and allows a higher-level workflow without introducing significant overhead compared to manual TCP management.

The second analysis evaluates the effect of the framework on DPU operation mapping. The DPU utilization results are reported in Table V. For ParallelExpertsNet, naive splitting reduces DPU utilization to 50–56%, since some boundary operations are excluded from accelerator mapping. After applying the framework, utilization increases up to 90.9%, confirming that correcting the boundary quantization context restores the expected mapping.

TABLE V
DPU UTILIZATION BEFORE AND AFTER APPLYING THE FRAMEWORK.

Arch.	Variant	Without (%)	With (%)
CNN	Small	50.0	84.6
CNN	Medium	50.0	84.6
CNN	Large	56.0	84.6
CNN	Huge	56.0	90.9
ConvViT	Small	4.2	95.6
ConvViT	Medium	4.2	97.6
ConvViT	Large	4.2	86.1

The ConvViT models show a different issue. Without the framework, DPU utilization remains at 4.2% for all variants, indicating that naive splitting generates fragments that are not sufficiently complete from the compiler's point of view. In this case, the problem is not limited to `fix` nodes at the boundaries, but also involves the collection of internal dependencies introduced by residual connections and aggregation operations. With the framework, dependency closure reconstructs structurally consistent fragments, increasing DPU utilization up to 97.6% for the Medium variant.

The latency impact for ParallelExpertsNet is shown in Fig. 2. The logarithmic scale highlights how naive splitting makes execution dominated by residual CPU operations, producing latencies up to two orders of magnitude higher than the corrected version. The most critical case is the Huge variant, which goes from 80,349 ms without the framework to 322.89 ms after correction, corresponding to a ×249 slowdown. The degradation is also significant for intermediate models, with slowdowns of ×162 for Medium and ×150 for Large.

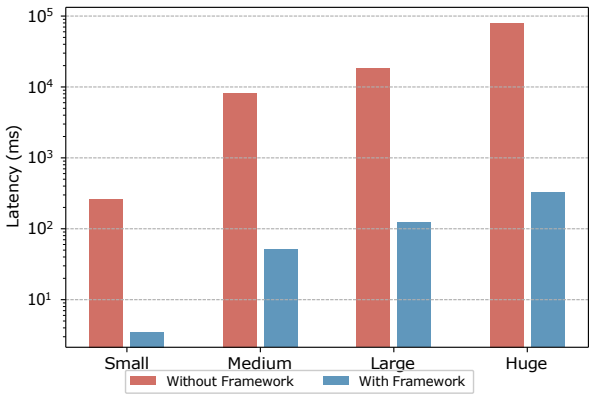

Fig. 2. ParallelExpertsNet inference latency before and after applying the framework, shown on a logarithmic scale.

The latency impact for ConvViT models is reported in Fig. 3. In this case as well, naive splitting produces a strong performance degradation, but the main cause is related to missing structural dependency collection in blocks with residual connections. The Small variant goes from 451 ms without the framework to 4.79 ms after correction, with a slowdown of about ×94, while the Medium variant goes from 3,521 ms to 39.37 ms. The most critical case is the Large variant, which goes from 16,997 ms to 31.36 ms, with a slowdown up to about ×542.

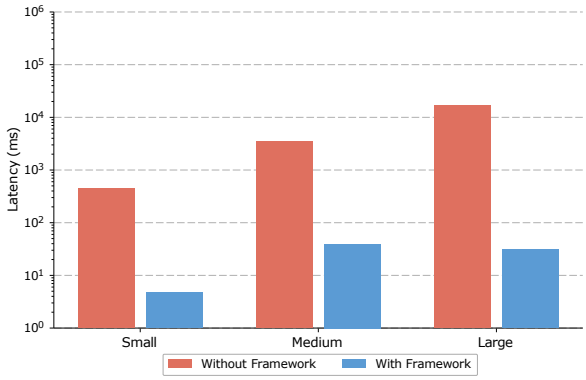

Fig. 3. ConvViT inference latency before and after applying the framework, shown on a logarithmic scale.

Overall, these results confirm that naive splitting can significantly degrade fragment compilation, but for different reasons depending on the model structure. In CNN models, the main issue is the loss of boundary quantization context, while in ConvViT models the dominant issue is the missing inclusion of dependencies generated by residual connections and aggregation operations. The proposed framework addresses both aspects, restoring effective DPU mapping and bringing latencies back to the expected range for hardware-accelerated execution.

The improvement in accelerator mapping introduces an offline preparation cost, reported in Table VI. This cost corresponds to the complete pipeline required to obtain `.xmodel` fragments suitable for deployment, and includes calibration,

quantization, splitting, and compilation. The evaluated configurations cover four different model sizes, ranging from approximately 1M to approximately 400M parameters.

TABLE VI
FRAMEWORK EXECUTION TIME IN SECONDS AS A FUNCTION OF MODEL SIZE.

Stage	~1M params	~25M params	~100M params	~400M params
Calibration	6.39	28.13	82.29	163.95
Quantization	0.41	2.29	11.70	32.51
Splitting	0.08	0.76	5.37	20.26
Compilation	0.95	6.26	36.01	247.69
Total	**7.83**	**37.44**	**135.37**	**464.41**

As shown in the table, the preparation time increases with model size. The complete pipeline requires 7.83 s for a small model with approximately 1M parameters and reaches 464.41 s for the largest model with approximately 400M parameters. The custom splitting stage remains limited compared with the other stages, requiring only 20.26 s even in the largest case, while calibration requires 163.95 s and compilation requires 247.69 s. This shows that the overhead introduced by the splitting step is limited compared with the overall cost of the Vitis AI pipeline.

VI. CONCLUSIONS AND FUTURE WORK

This work presented an XIR-level framework for splitting neural networks into independently compiled fragments while preserving effective DPU mapping. The empirical analysis of the Vitis AI compiler showed that naive splitting can remove quantization and structural information required for optimized compilation, leading to CPU fallback and significant latency degradation.

The proposed framework operates on the quantized XIR graph before compilation. Starting from a complete model, a calibration dataset, and user-defined cut points, it generates independent `.xmodel` fragments by reconstructing the quantization context through `fix` nodes and completing the structural dependencies required by each fragment. This allows the optimizations of the original model to be preserved without modifying the DPU or introducing additional hardware logic.

Experimental results on ParallelExpertsNet and ConvViT show that naive splitting can produce slowdowns up to ×249 on CNN-based models and up to about ×542 on ConvViT models. In both cases, the framework restores effective DPU mapping and brings latency back to the expected range for hardware-accelerated execution. The comparison between manual TCP execution and SLURM execution also confirms that the generated fragments can be deployed with standard tools while maintaining a high-level workflow.

Future work will focus on making the framework fully autonomous. A first direction is the automatic selection of cut points, so that the graph can be partitioned to maximize parallelism while preserving compiler optimizations. Another objective is to improve the distribution of fragments across the available computational nodes, avoiding unbalanced executions in which most of the workload is assigned to a single node. In this way, the framework could evolve from a user-guided splitting pipeline into an automatic partitioning and deployment tool for distributed DPU-based inference.

REFERENCES

[1] F. Buccellato, E. Vacca, S. Azimi, C. De Sio, and L. Sterpone, "From Detection to Intervention: An End-to-End System for Recognizing the "Signal for Help" Gesture in Real-Time," *Intelligent Systems with Applications*, vol. 26, Art. no. 200536, 2025, doi: 10.1016/j.iswa.2025.200536.

[2] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large Scale Distributed Deep Networks," in *Advances in Neural Information Processing Systems*, vol. 25, 2012, pp. 1232–1240.

[3] W. Jiang, E. H.-M. Sha, X. Zhang, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, "Achieving Super-Linear Speedup across Multi-FPGA for Real-Time DNN Inference," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 5s, pp. 67:1–67:23, Oct. 2019, doi: 10.1145/3358186.

[4] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[5] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[6] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks," in *Proc. 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, Monterey, CA, USA, Feb. 2015, pp. 161–170, doi: 10.1145/2684746.2689060.

[7] G. Cora, E. Vacca, C. De Sio, S. Azimi, and L. Sterpone, "Fast SEU Detection and Recovery in FPGA-Based AI Accelerators," *ACM Transactions on Reconfigurable Technology and Systems*, 2026, doi: 10.1145/3806052.

[8] G. Cora, D. Rizzieri, C. De Sio, S. Azimi, and L. Sterpone, "In-Hardware Fault-Tolerance Controller for Multi-FPGA Clustered Architectures," *IEEE Transactions on Computers*, 2026, doi: 10.1109/TC.2026.3709831.

[9] Y. Zhu, Z. He, W. Jiang, K. Zeng, J. Zhou, and G. Alonso, "Distributed Recommendation Inference on FPGA Clusters," in *Proc. 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 279–285.

[10] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. A. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 65–74.

[11] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From High-Level Deep Neural Models to FPGAs," in *Proc. 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 17:1–17:12.

[12] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A Toolflow for Mapping Diverse Convolutional Neural Networks on Embedded FPGAs," *arXiv preprint arXiv:1711.08740*, 2017.

[13] AMD, "Deep Learning Processing Unit," in *Vitis AI User Guide (UG1414)*, Version 3.5, Sep. 28, 2023. [Online]. Available: https://docs.amd.com/r/en-US/ug1414-vitis-ai

[14] AMD, "Vitis AI User Guide (UG1414)," Version 2.5, 2022. [Online]. Available: https://docs.amd.com/r/2.5-English/ug1414-vitis-ai

[15] F. Buccellato, C. De Sio, S. Azimi, and L. Sterpone, "On-Hardware Resilience Analysis of DPU-Accelerated CNNs on FPGA-Based Systems," in *Proc. 28th Euromicro Conference on Digital System Design (DSD)*, Salerno, Italy, 2025, pp. 34–41, doi: 10.1109/DSD67783.2025.00017.

[16] F. Buccellato, C. De Sio, S. Azimi, and L. Sterpone, "Hardware-Aware Runtime Detection of Soft-Error Anomalies in DPU-Accelerated Neural Networks," in *Proc. IEEE International On-Line Testing Symposium (IOLTS)*, 2026.

[17] AMD, "Xilinx Intermediate Representation," in *Vitis AI User Guide (UG1414)*, Version 3.5, Sep. 28, 2023. [Online]. Available: https://docs.amd.com/r/en-US/ug1414-vitis-ai/XIR

[18] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer, 2003, pp. 44–60, doi: 10.1007/10968987_3.

[19] A. B. Kahn, "Topological Sorting of Large Networks," *Commun. ACM*, vol. 5, no. 11, pp. 558–562, 1962, doi: 10.1145/368996.369025.