TISA: Tensor Instruction Set Architecture (Architecture Overview)

Vrushang Anand, Amogh Thiagarajan, Steven Kuzhipala
Department of Electrical Engineering and Computer Science
University of California, Irvine

Abstract
In recent years, the need for hardware accelerators has exploded in the fields of machine learning and artificial intelligence. Inference engines for many machine learning applications run on FPGAs due to their highly parallelized architectures and ultra-low latency. While most models are trained using high-level languages such as Python, running inference for these models optimally on FPGAs becomes very difficult due to the need for knowledge of RTL design. To address this challenge and eliminate the need to master RTL design, we introduce TISA, a universal architecture for ML/DL accelerators that is designed to use through high-level Python libraries such as PyTorch.

1 Introduction

The rapid growth of machine learning and deep learning has created increasing demand for computing platforms capable of executing neural-network inference with high throughput, low latency, and efficient power consumption. Graphics processing units have become the dominant platform for training and large-scale inference because of their programmability and high computational throughput. However, applications such as autonomous systems, robotics, real-time control, edge computing, and safety-critical systems often require predictable execution latency and hardware tailored to a specific computational workload. Field-programmable gate arrays provide an attractive platform for these applications because their configurable logic, on-chip memory, and parallel datapaths allow neural-network operations to be implemented as specialized hardware pipelines.

Despite these advantages, deploying neural networks on FPGAs remains difficult. Machine-learning models are generally developed using high-level frameworks such as PyTorch, while FPGA accelerators are traditionally implemented using hardware-description languages such as Verilog or VHDL. Mapping a model from a high-level computational graph to an efficient FPGA implementation requires knowledge of register-transfer-level design, memory organization, fixed-point arithmetic, pipelining, and the resource constraints of the target device. In many existing workflows, significant parts of the accelerator must also be redesigned or resynthesized when the model architecture or target hardware changes. This creates a substantial gap between machine-learning development and FPGA-based deployment.

A programmable neural-network accelerator can reduce this gap by separating the hardware architecture from the model being executed. Instead of generating an entirely new hardware design for every network, a reusable accelerator can be synthesized onto the FPGA once and subsequently programmed using a sequence of specialized instructions. The compiler is then responsible for translating model operations into instructions, selecting data layouts, partitioning large operations into hardware-compatible tiles, scheduling data movement, and adapting execution to the resources available on the target FPGA.

This report introduces the Tensor Instruction Set Architecture (TISA), a proposed programmable, target-parameterized architecture for FPGA-based machine-learning and deep-learning inference. TISA is designed around three decoupled execution engines: a load engine that transfers operands from external memory into on-chip memory, a compute

engine that performs tensor and activation operations, and a store engine that writes results back to memory. These engines operate concurrently and communicate through shared buffers, allowing data movement to overlap with computation. A specialized instruction set provides the interface between the compiler and the accelerator, enabling neural-network models to be executed without requiring the application developer to directly modify the underlying RTL implementation.

The long-term objective of TISA is to provide a software workflow in which models expressed using high-level frameworks such as PyTorch can be lowered into an intermediate representation, compiled into TISA instructions, and executed on a previously synthesized FPGA accelerator. The architecture is intended to remain consistent across FPGA platforms, while target-specific compiler parameters describe properties such as available compute units, on-chip memory capacity, external-memory bandwidth, and supported numerical precision. This approach allows the compiler to adapt operation tiling and scheduling without changing the high-level programming interface.

The initial prototype targets the AMD Kria KR260 platform and focuses on integer neural-network inference for a small fully connected model. The purpose of the prototype is not to compete with highly optimized commercial inference frameworks, but to validate the programmability of the architecture, the interaction between its load, compute, and store engines, and the feasibility of compiling high-level neural-network operations into accelerator instructions.

The primary contributions described in this report are:

1. The design of a programmable instruction-set architecture specialized for neural-network inference.

2. A decoupled load–compute–store microarchitecture that supports concurrent data movement and computation.

3. A compiler workflow for lowering high-level tensor operations into target-aware TISA instructions.

The remainder of this report describes the design requirements of TISA, the instruction-set and microarchitectural organization, the compiler workflow, the FPGA implementation, and the methodology used to evaluate the prototype.

2 System Architecture

TISA is a hardware-software co-design system that translates a trained neural network into an executable instruction stream and corresponding memory stream for an FPGA-based accelerator. The system is divided into two primary components: a host-computer side compiler and a programmable accelerator on the FPGA. The compiler lowers the model written in a high level language such as Python into hardware-compatible instructions, which are then executed by the accelerator.

2.1 Host side Compiler

The host-side compiler translates a trained ONNX model into a machine-oriented representation that can be executed by the TISA accelerator. Rather than lowering the model directly into binary instructions, the compiler uses multiple intermediate representations. Each representation removes one level of abstraction: the parser reconstructs the model graph, semantic analysis resolves its dimensions, TIR expresses its mathematical operations, and MIR maps those operations onto the accelerator's tiles and buffers.

Compilation is controlled by a target-specific MachineModel. This description contains architectural parameters such as the fixed-point format, machine word size, tile dimensions, and number of hardware buffers. These parameters allow the same compiler pipeline to generate different machine-level programs for different accelerator configurations without changing the original ONNX model.

2.1.1 ONNX Parsing

The compiler begins by reading the ONNX model as a binary byte sequence. Because these models are encoded using Protocol Buffers, the parser interprets the underlying protobuf wire format and reconstructs only the fields needed by the current compiler. These include graph nodes, model inputs, tensor dimensions, attributes, and initializer tensors containing weights and biases. Unsupported or irrelevant protobuf fields are skipped, allowing the parser to process ONNX files without reproducing the complete specification.

The result of this stage is an internal graph representation containing the model's operators, inputs, and constant tensors. At this point, the graph is a direct transcription of the file; no hardware resources, execution schedule, or memory locations have been assigned.

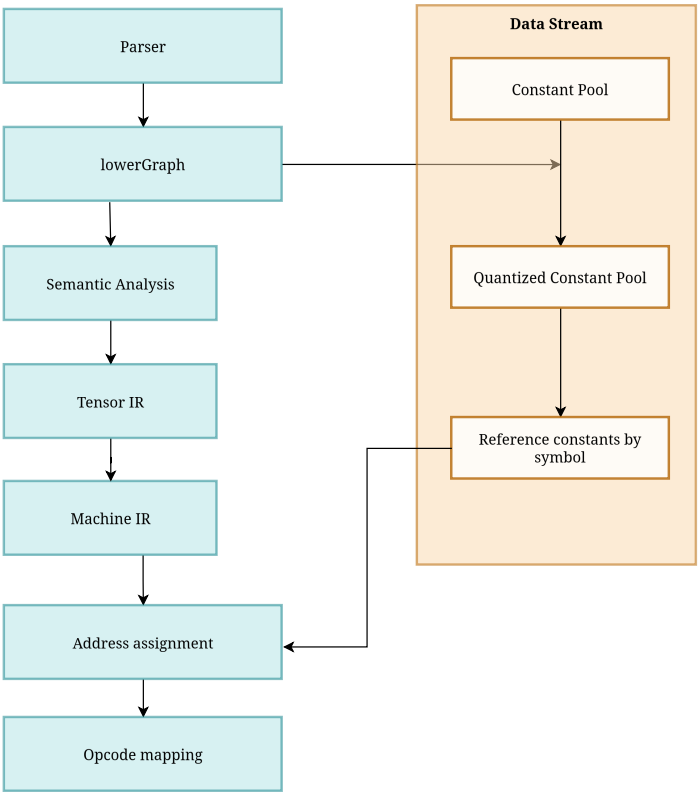

Figure 1: Compiler architecture

2.1.2 Graph Lowering and Constant Extraction

The decoded graph is next interpreted as a sequence of neural-network layers. The current implementation recognizes fully connected layers represented by ONNX Gemm operations and combines each operation with a following activation node when present. For example, a Gemm followed by Relu is represented as one layer containing a matrix multiplication, bias, and ReLU activation.

This stage also identifies the runtime network input and distinguishes it from initializer tensors. Weight layouts are normalized into the orientation expected by the compiler, including transposition when required by the ONNX Gemm attributes.

The lowering process creates two related outputs. The first is a structural representation containing the network topology, tensor names, dimensions, and activation types. The second is a constant pool containing the numerical weights and biases. This creates a separation between the model's computational structure and its parameter data.

2.1.3 Fixed-Point Quantization

TISA currently targets fixed-point inference. During graph lowering, floating-point initializer tensors are converted into the numerical format specified by the MachineModel. Each value is scaled according to the configured number of fractional bits and rounded to an integer representation.

Quantization failures are treated as compilation errors when a value cannot be represented by the selected word width. The compiler does not silently clamp overflowing values because saturation at this stage could produce an executable model with significantly altered behavior. Shared constants are deduplicated so that each initializer is quantized and stored only once.

After this stage, weights and biases reside in a constant pool as fixed-point integers. The structural compiler representations continue to refer to these constants symbolically by names such as @w1 or @b1, rather than embedding their numerical contents into individual operations.

2.1.4 Semantic Analysis

Semantic analysis validates the structural representation and converts partially specified tensor infor-

mation into a fully resolved network. It checks that tensor ranks and dimensions are valid, verifies that every referenced weight and bias exists in the constant pool, validates activation functions, and detects duplicate declarations.

Its primary responsibility is reconciling tensor sizes across layer boundaries. Each boundary may be constrained by several sources, including the preceding layer's output size, the following layer's input width, and the corresponding bias dimension. Semantic analysis requires these values to agree. A mismatch is reported before machine-specific lowering begins, preventing an invalid model from reaching the accelerator backend.

The resulting ResolvedNetwork contains a concrete input size and a sequence of resolved layers. Each layer has explicit input and output sizes, an activation enumeration, and symbolic references to its associated weights and bias.

2.1.5 Tensor Intermediate Representation

The resolved network is lowered into the Tensor Intermediate Representation. TIR describes the model as primitive operations over complete tensors using static single-assignment semantics. Each operation produces a new logical value, and later operations refer to that value through its identifier.

A fully connected layer is expanded into constant references, a matrix multiplication, a bias addition, and an optional activation operation. Thus, a high-level layer becomes a sequence similar to

$$y = \mathrm{ReLU}(Wx + b).$$

TIR captures the mathematical behavior and data dependencies of the model, but remains independent of the accelerator's physical organization. It does not yet contain tiles, hardware buffer identifiers, memory addresses, or encoded instructions. This separation allows graph-level transformations to be performed without reasoning about low-level machine details.

2.1.6 Machine Intermediate Representation

The Machine Intermediate Representation maps the whole-tensor operations in TIR onto the resources described by the MachineModel. This is the first stage where the compiler explicitly reasons about tile sizes, operand buffers, accumulation buffers, scratch storage, and the execution behavior of the accelerator.

Matrix operations larger than the hardware datapath are divided across output and reduction dimensions. For each output tile, MIR initializes an accumulation buffer and emits a sequence of tiled input loads, weight loads, and matrix multiply–accumulate operations. After all reduction tiles have been processed, the compiler emits the bias addition, an optional element-wise activation, and a store operation for the completed output tile.

The compiler alternates between available input and weight buffers so that data movement may overlap with computation. Accumulation buffers may similarly be alternated across output tiles, allowing one result to be stored while another is being computed. Intermediate layer outputs are assigned to reusable scratch regions. Because the current network representation is a linear chain, two scratch regions are sufficient for alternating between the previous and current layer outputs.

Edge tiles retain their actual dimensions when a tensor size is not evenly divisible by the configured tile size. The hardware can therefore execute a partially filled final tile without requiring the compiler to permanently pad or restructure the original tensor.

MIR memory references remain symbolic. Each reference contains a tensor or scratch symbol, an element offset, tile dimensions, and a row stride. For example, a tiled load may refer to a rectangular region of @w1 without yet assigning @w1 a physical DRAM base address. Keeping these references symbolic separates operation scheduling from the final memory-layout decision.

Element-wise operations such as ReLU can be applied as part of a tiled layer epilogue. Softmax is handled separately because it requires a reduction across an entire output vector and therefore cannot

generally be evaluated independently on each tile.

2.1.7 Memory Layout and Address Assignment

The MIR generated by the compiler contains explicit load, store, and compute operations, but its memory references remain symbolic. A weight tile may refer to a location such as @w1[offset], while an intermediate activation may refer to %scratch0[offset]. Before these operations can be encoded as TISA instructions, each symbolic reference must be assigned a concrete address in external memory.

The address-assignment pass consumes the symbolic MirProgram, the quantized constant pool produced during graph lowering, and memory-layout parameters supplied by the MachineModel. These parameters include the machine word size, alignment requirements, and the starting locations or sizes of the constant, runtime-input, output, and scratch-memory regions.

The pass distinguishes between three classes of memory objects. Constant symbols, including weights and biases, correspond to read-only tensors whose values are available at compile time. Runtime-input symbols identify tensors that are supplied by the host before inference begins and therefore do not appear in the constant pool. Scratch symbols represent temporary storage used for intermediate layer outputs. Scratch regions require reserved memory capacity but no initial data.

Memory is allocated using a deterministic linear layout. For each region, the compiler maintains a cursor representing the next available location. Each tensor is assigned the current cursor value as its base address, after applying any required alignment, and the cursor is advanced by the number of machine words occupied by the tensor. Processing symbols in a fixed order ensures that identical models and machine configurations produce reproducible addresses, which simplifies testing and binary comparison.

The result of this process is a symbol-to-address map of the form [base(symbol) → DRAM word address]

Each MIR memory reference is then resolved using

base(symbol) + offset

The tile dimensions and row stride stored in the original memory reference are preserved. Consequently, a symbolic reference such as

[`@w1[20, 4 × 4, stride 5]`]

may become

[`DRAM[20], 4 x 4, stride 5.`]

The address identifies the first element of the tile, while the dimensions and stride describe how the remaining elements are arranged. Whether this description is encoded directly in a memory instruction or expanded into multiple contiguous transfers is determined by the final ISA design.

Address assignment is also the point at which the compiler's structural and data streams rejoin. Until this stage, MIR instructions refer to parameters only by symbol, while the constant pool separately contains their quantized values. The memory-layout pass allocates a flat DRAM data image and copies each constant tensor into the region assigned to its symbol. Runtime-input and scratch regions are reserved within the same memory image but are initially empty or zero-filled.

The output of this stage is therefore composed of two related artifacts:

1. Located MIR, in which symbolic memory references have been replaced by concrete DRAM addresses.

2. A DRAM data image, containing the quantized constant pool together with reserved runtime-input, output, and scratch regions.

The layout algorithm is independent of a particular FPGA board. Retargeting the program primarily changes the machine configuration, memory-region sizes, alignment rules, and resulting addresses rather than the structure of the pass itself.

2.1.8 Opcode Selection and Instruction Encoding

Once all memory references have concrete addresses, the located MIR can be translated into the binary instruction format understood by the FPGA instruction decoder. This stage maps each machine-level operation to an ISA opcode, encodes

its operands into defined bit fields, and packs the result into one or more instruction words.

Opcode selection is performed through a mapping between each MirOp and its corresponding TISA opcode. Operations such as tiled loads, tiled stores, matrix multiplication, addition, accumulator initialization, activation, and softmax each require a unique encoding. This mapping provides a direct boundary between the compiler's internal machine representation and the externally visible instruction-set definition.

The encoder next translates MIR operands into hardware-defined field values. Logical input, weight, and accumulation-buffer references are converted into numeric buffer selectors. Resolved DRAM addresses are inserted into immediate or address fields. Additional fields may describe tile dimensions, row stride, shift amounts, activation modes, or synchronization behavior.

For example, a conceptual instruction format may contain fields for an opcode, destination buffer, source buffers, control flags, and an immediate value. The exact partitioning of these fields depends on the final instruction width and on which values must be selectable at runtime. Fixed-point scaling may either be represented explicitly through shift fields or omitted when the configured fractional precision is hardwired into the accelerator.

Instruction width is an important architectural constraint. External-memory addresses, buffer selectors, accumulator identifiers, tile dimensions, and operation flags must all fit within the available encoding space. A compact instruction word may be sufficient for arithmetic operations but too small for memory-transfer instructions with full addresses and tensor descriptors. In that case, the ISA may use wider instructions, multiple instruction formats, or multiword encodings in which an opcode word is followed by address or descriptor words.

The encoder packs the selected opcode and operand fields into a sequence of fixed-width machine words. These words form the instruction image that is loaded into the accelerator's instruction memory. The compiler backend therefore produces the following final outputs:

1. An encoded TISA instruction image.
2. A DRAM data image containing constants and reserved runtime memory.
3. Metadata describing input and output addresses required by the host runtime.

The correctness of instruction encoding can be tested using an encode–decode–execute workflow. Generated instruction words are decoded using logic equivalent to the FPGA decoder and executed in a software model of the accelerator. The resulting output is then compared against the TIR or reference-model output. Agreement between the two verifies that opcode selection and bit-field packing preserve the semantics of the located MIR.

An optional instruction-scheduling pass may be inserted between address assignment and encoding. Such a pass would reorder independent load, compute, and store operations to expose overlap between the accelerator engines while preserving data dependencies. It would not change tensor addresses or operation semantics. If the hardware controller dynamically pipelines a sequential instruction stream, explicit compiler scheduling may not be necessary; otherwise, static scheduling can be used to make double buffering and engine-level concurrency visible in the final instruction order.

2.2 FPGA Execution Architecture

The TISA accelerator uses a decoupled access–execute architecture composed of three autonomous engines: a load engine, a compute engine, and a store engine. Each engine maintains its own program counter and executes a separate instruction stream. Rather than relying on a central sequencer to coordinate every operation, the engines synchronize through shared on-chip buffers and lightweight handshake signals.

The load engine transfers tensors from external DRAM into input and weight buffers, the compute engine performs tiled neural-network operations, and the store engine writes completed results back to DRAM. This separation allows memory reads, arithmetic operations, and memory writes to

execute concurrently when their data dependencies permit.

Conceptually, execution follows the path

[DRAM → Load Engine → Compute Engine → Store Engine → DRAM.]

Intermediate activations between layers are written to a scratch region in external memory by the store engine and subsequently loaded by the load engine for the next layer. Although this round trip introduces additional memory traffic, it simplifies synchronization and allows intermediate tensors larger than the available on-chip memory to be processed.

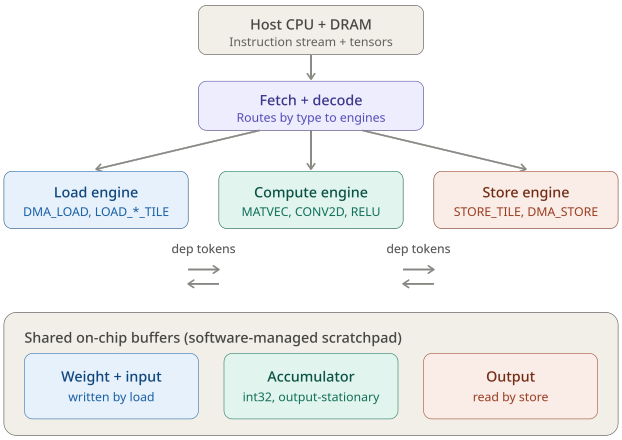

Figure 2: FPGA Accelerator Architecture

2.2.1 Ownership

The engines communicate through explicitly managed on-chip buffers. The initial design contains two input buffers, two weight buffers, and two accumulation buffers:

- IB0 and IB1 hold input-activation tiles.
- WB0 and WB1 hold weight tiles.
- ACC0 and ACC1 hold accumulated output tiles.

The input and weight buffers are produced by the load engine and consumed by the compute engine. The accumulation buffers are produced by the compute engine and consumed by the store engine.

Each pair of buffers forms a depth-two ping-pong channel. While one slot is being consumed, the producer may fill the other slot with data for a later operation. This allows the transfer of the next tile to overlap with the computation of the current tile.

Buffer dimensions are determined by the machine configuration. For a compute tile with output dimension T_M and reduction dimension T_K, an input buffer contains T_K elements, while a weight buffer contains $T_M T_K$ elements. An accumulation buffer contains T_M wide accumulator values.

2.2.2 Load Engine

The load engine moves rectangular tensor regions from external memory into the input and weight buffers. A load descriptor specifies the base DRAM address, tile dimensions, tensor row stride, and destination buffer.

For a tile with (R) rows, (C) columns, base address (B), and row stride (S), the load engine generates addresses according to

$A(r, c) = B + rS + c$

The engine first waits until the selected destination buffer is empty. It then uses a strided address generator to issue the required DRAM reads and places the returned elements into consecutive locations in the on-chip buffer. After the complete tile has been loaded, the engine marks the destination buffer as ready.

The load engine may run ahead of the compute engine by prefetching the next input and weight tiles into the unused ping-pong slots. Its performance therefore depends on both the external-memory interface and the compiler's ability to generate a stream that exposes sufficient overlap.

2.2.3 Compute Engine

The compute engine contains the primary arithmetic datapath of the accelerator. Its central operation is a tiled multiply–accumulate computation of the form

$$\mathbf{a} \leftarrow \mathbf{a} + \mathbf{W}\mathbf{x}$$

where (W) is a weight tile, (x) is an input tile, and (a) is an accumulation tile.

For a $T_M \times T_K$ datapath, the compute engine contains T_M output lanes. Each lane consumes one row of the weight tile and the shared input vector. The elements are multiplied in parallel, combined through an adder tree, and accumulated into the corresponding output element.

With a 4×4 configuration, for example, the engine contains four output lanes, each with four multipliers. A single compute step evaluates four partial dot products in parallel. Multiple reduction tiles are accumulated into the same accumulation buffer until the complete input dimension has been processed.

The compute engine waits until the selected input and weight buffers are ready. After consuming them, it clears their ready flags, allowing the load engine to reuse those slots. When processing the final reduction tile for an output block, the engine performs the layer epilogue, which may include bias addition and an activation function, before marking the selected accumulation buffer as ready for the store engine.

2.2.4 Store Engine

The store engine consumes completed accumulation tiles and writes them to external memory. A store descriptor identifies the source accumulation buffer, destination DRAM location, and number of valid output elements.

The engine waits until the selected accumulation buffer is marked ready. It then applies the required fixed-point narrowing and saturation to each valid element before issuing the memory writes. Once all elements have been stored, the engine clears the accumulation-ready flag, returning ownership of the buffer to the compute engine.

Partial output tiles are handled using their actual valid extent. Therefore, when the output dimension is not divisible by T_M, the store engine writes only the valid elements of the final tile.

2.2.5 Independent Instruction Streams

Each engine fetches instructions from a separate region of instruction memory and advances its own program counter. The load stream describes incoming tensor transfers, the compute stream describes arithmetic operations, and the store stream describes outgoing transfers.

This organization differs from a single sequential instruction stream in which one controller explicitly issues every load, compute, and store operation. In TISA, synchronization occurs through the shared-buffer protocol. An engine may advance independently until its next instruction depends on a buffer that is not yet available.

The compiler or a scheduling pass is responsible for generating compatible streams. The relative instruction order must ensure that every produced tile is eventually consumed and that every consumer waits on a tile that will eventually be produced.

2.2.6 Steady-state Pipelining

After the pipeline has been filled, all three engines may operate simultaneously. While the compute engine processes one tile, the load engine may fetch the next tile and the store engine may write a previously completed output.

A representative schedule is

	t_0	t_1	t_2	t_3
Load	(m_0, k_0)	(m_0, k_1)	(m_1, k_0)	(m_1, k_1)
Compute	—	(m_0, k_0)	(m_0, k_1)	(m_1, k_0)
Store	—	—	—	m_0

where m identifies an output tile and k identifies a reduction tile.

Without overlap, the time required for each tile would approximately equal the sum of its load, compute, and store times. Under steady-state overlap, the tile throughput instead approaches

$$T_{\text{tile}} \approx \max\left(T_{\text{load}}, T_{\text{compute}}, T_{\text{store}}\right).$$

The slowest engine therefore determines sustained throughput.

2.2.7 Ordering and Deadlock Considerations

The ready-flag protocol prevents common read-after-write and write-after-read hazards between adjacent

engines. A producer cannot overwrite a full slot, and a consumer cannot read an empty slot. Because each engine executes its local stream in program order, the protocol provides deterministic ownership of every shared buffer.

For a feed-forward layer execution schedule, the load–compute–store channels do not inherently form a cyclic dependency. Nevertheless, correctness depends on the instruction streams being generated consistently. A missing producer, an incorrect buffer selector, or incompatible stream lengths could leave one engine waiting indefinitely.

Layer boundaries introduce an additional dependency that is not represented directly by the on-chip buffer flags. When one layer stores an intermediate activation to DRAM, the next layer must not load that region before the store has completed. This dependency can be enforced through explicit barrier instructions, completion counters, stream-level ordering constraints, or a memory-dependency token associated with the scratch region.

2.2.8 FPGA Resource Mapping

The compute datapath is primarily mapped onto FPGA DSP blocks. On devices where a signed 32-bit multiplication requires multiple DSP slices, the total DSP usage grows approximately with the product $T_M T_K$. Tile dimensions therefore act as the principal scaling parameters for compute parallelism.

Input, weight, accumulation, and instruction memories are implemented using block RAM or distributed memory, depending on their required capacity and access pattern. External DRAM stores the model parameters, runtime inputs, outputs, and intermediate tensors that cannot remain on chip.

The MachineModel communicates these hardware constraints to the compiler. If a target device has fewer DSP slices or less on-chip memory, the tile dimensions and buffer capacities can be reduced. The compiler then automatically generates a larger number of smaller tiled operations while preserving the same model-level computation.

3 Next Steps

The next stage of TISA development will focus on introducing optimization passes across the compiler's intermediate representations. At the TIR level, these passes will simplify the computational graph, remove redundant operations, propagate constants where possible, and fuse compatible operations such as matrix multiplication, bias addition, and activation. At the MIR level, optimization will focus on hardware-aware transformations, including tile-size selection, reuse of scratch-memory regions, reduction of unnecessary DRAM transfers, improved ping-pong-buffer scheduling, and reordering of independent load, compute, and store instructions to increase engine overlap. These optimizations will use the target-specific MachineModel to account for the available DSP resources, on-chip memory capacity, buffer dimensions, numerical precision, and external-memory bandwidth of the selected FPGA configuration.

The first end-to-end validation target will be an MNIST fully connected network deployed on the AMD Kria KR260. A trained model will be exported to ONNX, parsed and lowered through the TISA compiler, quantized into the selected fixed-point format, divided into hardware-compatible tiles, and encoded into separate instruction streams for the load, compute, and store engines. The compiler will also generate the DRAM image containing the model parameters and reserved regions for inputs, intermediate activations, and outputs. These artifacts will then be transferred to the KR260 and executed using the synthesized TISA accelerator. The FPGA output will be compared against a software reference to verify numerical correctness, after which the prototype will be evaluated using inference latency, classification accuracy, resource utilization, and the degree of overlap achieved between data movement and computation.

4 Conclusion

TISA presents a programmable FPGA-based inference architecture that separates neural-network

models from the underlying RTL implementation. Its compiler lowers ONNX models through progressively lower-level representations, performs target-aware quantization, tiling, memory assignment, and instruction generation, and produces the instruction and data images required by the accelerator. On the FPGA, independent load, compute, and store engines communicate through shared ping-pong buffers, allowing data movement and computation to overlap while preserving deterministic execution. The initial KR260 prototype will focus on validating this end-to-end workflow for fixed-point fully connected inference, including compiler correctness, engine synchronization, FPGA execution, and comparison against a software reference. This prototype will establish the foundation for extending TISA with additional operators, improved scheduling, broader model support, and more advanced architectural optimizations.