SHARD: A Compatibility Framework for Deploying Transformer Models on Edge NPUs

Adhitya Mohan
adhitya.mohan@colorado.edu
University of Colorado Boulder
Boulder, Colorado, USA

Richard Thompson
rith1339@colorado.edu
University of Colorado Boulder
Boulder, Colorado, USA

Eric Keller
eric.keller@colorado.edu
University of Colorado Boulder
Boulder, Colorado, USA

Mark Zhao
myzhao@colorado.edu
University of Colorado Boulder
Boulder, Colorado, USA

Abstract

Running inference on diverse transformer models, especially for emerging architectures such as Vision-Language Models (VLMs), is often infeasible on many edge accelerators such as Neural Processing Units (NPUs). This limitation arises because transformer execution graphs frequently violate constraints imposed by vendor-provided, black-box NPU toolchains. These constraints include restricted operator sets, rigid tensor shape and layout requirements, and tight hardware limits such as limited on-chip SRAM. Consequently, many NPUs fail to execute modern transformer models end-to-end, or they fall back to CPU execution resulting in degraded performance and efficiency.

We present SHARD, a compatibility framework to maximize the deployability of diverse transformers on edge NPUs. SHARD transforms high-level models into portable input graphs (e.g., ONNX) compatible with diverse NPU hardware and toolchain constraints. SHARD introduces constraint-driven rewrites that decompose and explicitly shard execution graphs to fit operations within restricted on-chip SRAM limits, legalize unsupported operators by lowering them into hardware-supported primitives, and address impacts on model accuracy as a result of quantization. Using a Rockchip RK3588 NPU, we demonstrate how SHARD enables a vision encoder to natively execute on the NPU, achieving a 8.7× speedup over the vendor toolchain which relies on CPU fallback.

CCS Concepts: • **Computer systems organization → Embedded systems**; • **Computing methodologies → Neural networks**.

EuroMLSys '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2605-7/2026/04
https://doi.org/10.1145/3805621.3807618

Keywords: edge NPUs, transformer inference, model deployment, graph rewriting, embedded machine learning

ACM Reference Format:
Adhitya Mohan, Richard Thompson, Eric Keller, and Mark Zhao. 2026. SHARD: A Compatibility Framework for Deploying Transformer Models on Edge NPUs. In *Sixth European Workshop on Machine Learning and Systems (EuroMLSys '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3805621.3807618

1 Introduction

Edge applications, such as robotics [14], IoT [3], and healthcare [15], are increasingly reliant on transformer-based, multimodal foundation models. As these models move from the cloud to resource-constrained edge devices, developers are increasingly deploying them on specialized hardware accelerators, including Neural Processing Units (NPUs) such as Qualcomm Hexagon [25], Google EdgeTPU [5], Rockchip NPU [30] and others [1, 21]. These NPUs employ specialized hardware to perform energy-efficient tensor operations to provide real-time inference under limited power budgets.

However, modern NPUs rely on vendor-specific SDKs and toolchains rather than open machine learning compilation frameworks [2, 4, 22, 33]. Examples include SNPE [23], RKNN [28], and Android NNAPI [11]. These toolchains are a black box: they accept a model graph (e.g., ONNX) and return a binary artifact. Developers perform targeted graph rewrites that satisfy hard constraints on supported operators, tensor shapes, alignment rules, fusion behavior, and control-path limits [26, 29].

Deploying transformers on NPUs is challenging because many NPUs and vendor ecosystems were historically designed for prior models such as CNNs [7, 12]. In contrast, transformer models introduce distinct execution requirements, including attention primitives, dynamic tensor shapes, and memory-intensive intermediates that are not well supported by existing edge runtimes. When an edge toolchain encounters an unsupported operator or constraint (e.g., size) violation, it falls back to *hybrid execution*: running supported portions of the model on the NPU and the rest on the CPU.

While functionally correct, the overhead of context switching, driver synchronization, and pipeline disruption can largely negate the NPU's benefit.

In this paper, we address these limitations by maximizing the *deployability*, and thus performance, of emerging transformer-based foundation models [10, 35] on modern edge NPUs. We argue for a framework that systematically transforms transformer graphs to satisfy the hard deployability constraints imposed by black-box, vendor-specific NPU toolchains. Such a framework must be *a) NPU-agnostic* across diverse vendors and devices, *b) backwards-compatible* across changing SDK versions, *c) model-independent* across a broad range of models, and *d) fidelity-preserving* so that model accuracy does not degrade.

Towards these goals, we introduce SHARD, a compatibility framework for deploying transformer models on edge NPUs. SHARD accepts a portable model graph (e.g., PyTorch), alongside a *compatibility profile* describing the constraints of a target, vendor-specific toolchain. Constraint-driven passes then rewrite the graph into several compilation units (*shards*) that are natively compatible with the vendor toolchain. SHARD uses the vendor toolchain to produce a *shard pack* – a set of binaries for each shard. Finally, a lightweight runtime executes shards sequentially and runs the end-to-end model.

We evaluate a prototype of SHARD on a Rockchip RK3588 NPU using a modern VLM (SmolVLM-256M). This model cannot natively run on the RKNN toolchain and resorts to hybrid execution. Using SHARD, we natively support the VLM on Rockchip's RK3588 NPU, improving inference latency by 8.7× and fidelity (0.95 versus 0.64 cosine similarity to the ground truth) compared to the native RKNN toolchain.

2 Background and Motivation

Edge computing increasingly relies on specialized NPUs (e.g., Qualcomm Hexagon [25], Google EdgeTPU [5], Rockchip NPU [30]) for low-latency, on-device inference. However, these devices require the use of proprietary software stacks (e.g., EdgeTPU [6], Hexagon SDK [24], RKNN-Toolkit [28]).

These opaque toolchains impose numerous constraints that limit model deployment. *Control-path limits:* Large graphs create monolithic binaries that crash device drivers during parsing (see Section 4). *Data-path limits:* On-chip SRAM limitations impose constraints on local working-set sizes (e.g., 16K element transpose limits or 32KB L1 SRAM) [27]. *Unsupported Operators:* Complex non-linearities (e.g., GELU) and normalizations (e.g., LayerNorm) are often absent from proprietary libraries [26, 29]. This can trigger compilation failure, CPU fallback, or execution in the wrong precision (e.g., INT8 instead of FP16). *Numerical range/precision mismatch:* Outliers in reduced precision paths (INT8/FP16) cause fidelity collapse [9, 17, 34], impacting model accuracy. *Fusion Hazards:* Finally, compilers can apply opaque operator fusion heuristics, which can reintroduce unsupported patterns or alter numerical precision.

Furthermore, NPUs and their compilers were optimized for static models such as CNNs [12]. These models have predictable memory access and bounded activations, so edge NPUs pair massive Multiply-Accumulate (MAC) arrays with minimal on-chip SRAM and slower external DRAM. However, transformers introduce computational paradigms hostile to these constraints. Self-attention materializes an $L \times L$ intermediate matrix for sequence length L. This quadratic memory complexity exhausts limited SRAM, causing DRAM thrashing or compilation failure. Transformers also introduce new operators (e.g., large transposes, complex nonlinearities) absent from existing toolchain libraries [7, 26, 29].

While open compiler frameworks like TVM [4], IREE [22], Torch MLIR [18], and torch.compile [2] provide robust, extensible infrastructure for deep learning compilation, they are often insufficient for edge NPUs. These frameworks rely on open-source drivers and detailed hardware specifications to generate efficient machine code. However, edge NPU vendors typically provide closed-source, proprietary toolchains (e.g., Rockchip's RKNN) and withhold the low-level ISA documentation required to build a custom backend. Instead, developers must manipulate the ONNX graph *before* compilation, requiring extensive manual effort to create device and model-specific conversion pipelines to utilize NPUs.

In summary, edge NPUs provide limited support for transformer based models, often resulting in reduced performance and inefficient execution due to hybrid NPU-CPU fallback. Moreover, black-box vendor-specific toolchains require developers to repeatedly adapt models for each NPU and model combination, significantly increasing engineering effort as transformer architectures continue to evolve. Unlike existing ML compiler frameworks, which provide transparent and configurable toolchains, edge NPUs toolchains are largely opaque, leaving the input model graph as the primary mechanism for expressing hardware compatibility. This creates a need for a framework that decouples model compatibility from vendor-specific toolchains, enabling transformers to be deployed efficiently across a broad portfolio of edge NPUs.

3 The SHARD Framework

To meet these goals, we propose SHARD, a model compatibility framework for modern NPUs. Figure 1 shows the end-to-end workflow of SHARD.

System Inputs. SHARD takes as input a model checkpoint, such as a PyTorch model downloaded from an open-source repository like Hugging Face. SHARD then exports the model to a static ONNX graph with PyTorch's built-in ONNX exporter, `torch.onnx.export`. Users also provide SHARD with

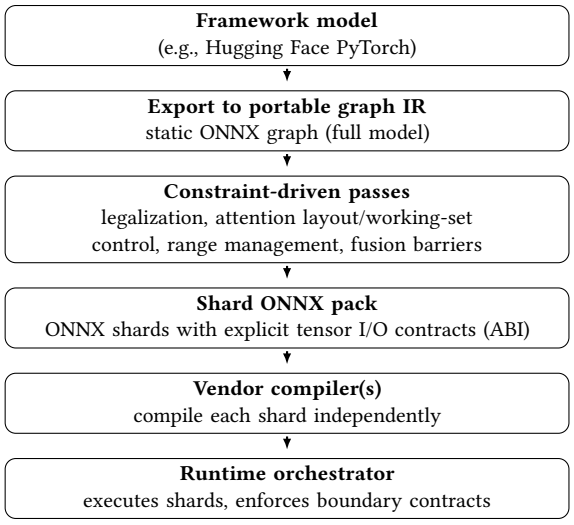

Figure 1. SHARD as a deployability-first pre-compiler. A framework model is exported to ONNX, rewritten into shards guided by a compatibility profile, compiled shard-by-shard by a black-box vendor toolchain, and executed under an explicit runtime contract.

a *compatibility profile*. The profile specifies a) supported operators, b) allowed tensor ranks, layouts, and dtypes, c) hardware limits on artifact sizes, and d) precision requirements (e.g., FP16 vs INT8) for regions of the graph.

Constraint-driven passes. SHARD next applies a series of constraint-driven transformation passes to the ONNX graph. Each pass checks whether portions of the graph violate constraints specified in the compatibility profile. When violations are detected, the pass applies graph rewrites that transform unsupported patterns into equivalent forms that satisfy the target constraints. These passes operate directly on ONNX semantics and preserve functional equivalence.

Shards. During rewriting, SHARD partitions the ONNX graph into disjoint subgraphs called *shards*. Shards serve two purposes. First, smaller compilation units improve compilation robustness because large graphs often result in failures due to compiler timeouts and on-chip SRAM limitations. Secondly, shards create a predictable working set and establishes explicit tensor I/O boundaries that compilers cannot cross-fuse, which may violate compatibility assumptions introduced by earlier rewrites. SHARD then invokes the NPU-specific compiler to generate executable binaries for each shard.

Runtime. Finally, SHARD aggregates shard binaries into a "shard pack" that collectively implements end-to-end inference. SHARD enforces a strict binary interface between each shard by specifying fixed tensor I/O formats, explicit materialization points, and a well-defined execution order. At runtime, a lightweight orchestrator schedules shard execution on the NPU and manages intermediate tensors, enabling the full model to execute using independently compiled shards.

3.1 Constraint-Driven Passes

We now describe the constraint-driven passes. These passes do not replace vendor compilers. Instead, they rewrite a model into a form that the target toolchain can compile and run, guided by the input compatibility profile. Because closed vendor compilers often lack dynamic control flow support and dynamic memory paging, SHARD begins by flattening the the model graph (e.g., matrix multiplication loops) into discrete, unrolled sequences of operations. SHARD next performs passes to (1) satisfy control-path size constraints, (2) satisfy data-path size constraints, (3) legalize operators, and (4) enforce precision constraints. These passes address the key constraints imposed by opaque NPU toolchains, as specified in Section 2.

3.1.1 Enforcing control-path constraints. SHARD turns a monolithic transformer graph into a *shard pack*: a sequence of independently compilable units with typed tensor I/O. SHARD does this because the exported/unrolled static graph can hit practical toolchain and device limits due to artifact size, resulting in compiler timeouts and driver crashes.

SHARD shards each transformer layer to satisfy graph size constraints specified in the compatibility profile. For example, SHARD splits each transformer layer into an attention shard and an MLP shard. The split keeps graphs smaller and enables targeted rewrites. More sub-sharding is possible when one shard still exceeds profile limits. For example, an attention shard can be further cut into (i) QKV projections, (ii) score matmul + softmax, (iii) value matmul, and (iv) output projection, or split by sequence tiles; each cut reduces peak working set and constrains compiler rewrites inside a smaller region.

SHARD also adds a well-defined interface around each shard, which allows each shard to seamlessly communicate with other shards. For example, our current prototypes specify the following.

- **Tensor conventions:** each shard has a fixed, typed tensor signature at its inputs/outputs (rank, layout, dtype).
- **Fixed maximum length:** the runtime enforces a maximum sequence length `MAX_LEN` via padding and cropping at shard boundaries.
- **Execution order:** shards execute sequentially in a fixed order specified by the shard pack; intermediate tensors are materialized at boundaries.

3.1.2 Enforcing data-path constraints. SHARD next addresses deployability issues introduced by attention, which can violate data-path limits of hardware (Section 2). Unlike traditional feed forward networks, attention mechanisms require large permutations and materialize large score matrices ($L \times L$) that overwhelm hardware and SRAM limits.

Given tensor size and memory constraints specified in the compatibility profile, SHARD addresses this with two simple, general transforms: *safe transpose* and *tiling*. Safe transpose replaces one large permutation with a sequence of smaller reshapes and transposes over bounded tiles. We also compute attention in tiles along the sequence dimension and concatenate outputs, so the backend never has to materialize a full $L \times L$ matrix. Together, these transforms keep both the permutation domain and the peak intermediates within size and memory limits. For example, the RK3588 has 32KB of scratchpad, resulting in a maximum safe tile of 16K FP16 elements. SHARD tiles the sequence length into blocks of 256×64, perfectly meeting the 32KB limit.

3.1.3 Legalizing operators. Modern transformers rely on complex non-linearities (e.g., GELU) and normalizations (e.g., LayerNorm) often unsupported by edge compilers [26, 29], triggering CPU fallback or forcing execution into the wrong precision (e.g., INT8 instead of FP16). Next, SHARD rewrites unsupported operators, or operators which violate precision constraints, into supported equivalents (or controlled approximations). We do so by systematically applying a series of rule-based transformations during this pass.

For example, GELU is rewritten as its fast approximation:

$$\mathrm{GELU}(x) \approx 0.5x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right)\right) \quad (1)$$

or as $x \cdot \sigma(1.702x)$. LayerNorm is decomposed into explicit mean, variance, and normalization steps. This forces the compiler to map the entire architecture to the NPU.

3.1.4 Enforcing precision constraints. Reduced precision operators commonly required by NPUs are numerically fragile: activations can overflow in FP16 or lose signals in INT8. This is problematic for Vision-Language Models, which exhibit massive activation spikes [9, 34]. Moreover, vendor compilers aggressively fuse operators across precision levels, which can violate precision constraints enforced by our operator legalization pass. For example, a compiler may fuse LayerNorm into an INT8-only operator even when the profile requires FP16 normalization. SHARD addresses these issues with two techniques.

First, SHARD performs a profiling pass to detect if a shard produces intermediates that may result in numerical saturation. We sweep a small set of candidate values and keep the smallest scale factor λ that avoids categorical failures (NaN/Inf, runtime crashes, or large drift) while preserving stable compilation. SHARD then statically scales activations by a scalar λ. Let z denote the activation tensor entering the shard pack (i.e., the boundary of the first shard). The orchestrator applies an inverse scale on entry to the pack and a forward scale on exit from the pack:

$$z_{\text{in}} = z/\lambda, \quad z_{\text{out}} = \lambda \cdot F(z_{\text{in}}) \quad (2)$$

where F denotes the composite function of all shards in the pack. Because F contains non-linear operators (e.g., GELU, Softmax), this transformation is *not* mathematically exact: $\lambda \cdot F(z/\lambda) \neq F(z)$ in general. Instead, it is a controlled approximation that trades a small fidelity cost for preventing catastrophic FP16 saturation. As shown in Table 3, this tradeoff is strongly favorable: with scaling, end-to-end cosine similarity remains > 0.95, whereas without scaling, fidelity collapses to 0.11 by Layer 5.

Secondly, if the vendor compiler may fuse two operators which violate the compatibility profile, SHARD introduces "fusion barriers" that force the compiler to break the fusion. The primary barrier is the shard boundary itself: by partitioning the graph into separate compilation units with explicit tensor materialization at boundaries, SHARD physically prevents cross-shard fusion. Within a shard, if further fusion control is needed, SHARD injects a lightweight non-linear dependency:

$$X_{out} = X + \epsilon \cdot \sigma(X_{slice}) \quad (3)$$

where ϵ and the activation σ are chosen to be platform-appropriate (e.g., large enough to survive reduced-precision rounding while remaining numerically negligible). This intra-shard gadget supplements the topological barrier; the specific ϵ value is tuned per platform and precision format to ensure it is not optimized away.

4 Implementation and Evaluation

We implemented a prototype of SHARD as a Python package. SHARD's constraint-driven passes are implemented on top of PyTorch operators. We use PyTorch to export ONNX graphs (shards) that are subsequently passed to the vendor compiler. Our prototype supports the Rockchip RKNN toolchain. We use the `rknn-toolkit2` for compilation, which produces .rknn binaries. We implement a lightweight runtime in Python using the `rknn-toolkit-lite2` library. The runtime executes shards in round robin across NPU cores to improve utilization while preserving the shard execution order.

4.1 Experimental Setup

Hardware and Software Platform. We evaluate SHARD on an Orange Pi 5 Max SBC [31], powered by the Rockchip RK3588 SoC. The SoC integrates an octa-core CPU (4×Cortex-A76 at 2.4GHz, 4×Cortex-A55 at 1.8GHz) and a tri-core NPU providing 6 TOPS of peak performance. The NPU uses a dedicated instruction set and architectural constraints that are not publicly documented, including a 32KB local SRAM limit (imposing a hard cap of 16,384 FP16 elements per tensor operation) and a 956KB global SRAM working set limit. The system runs Linux 6.1.115 (vendor kernel) with Python 3.10. Compilation is performed on an x86 host using the user-space `rknn-toolkit2`, and inference uses the on-device `rknn-toolkit-lite2` runtime.

Table 1. End-to-End performance and fidelity results.

Configuration	Latency (s)	Cosine Similarity
RKNN-INT8	1.40	0.02
RKNN-FP16	19.63	0.64
CPU-FP32	30.0	1.00
SHARD	2.24	0.95

Target Model. We target the 93M-parameter SigLIP-B/16 vision encoder of SmolVLM-256M [19]. This encoder is a difficult workload for edge NPUs: attention introduces large transposes and intermediate tensors, GELU stresses limited operator support, and activation outliers make reduced precision brittle. Unlike ResNet-style CNNs that align with common NPU assumptions such as NHWC layouts and ReLU activations, Vision Transformers routinely trigger control-path limits, CPU fallback, and destructive precision rewrites.

Baseline RKNN Behavior. Under default RKNN compilation, this workload exhibits two failure modes. The toolchain quantizes broad regions to INT8, which collapses fidelity, and it falls back to the CPU for large attention transposes that exceed hardware or compiler limits. A manual FP16 decomposition improves fidelity only modestly and still incurs high latency because the transpose remains too large to execute as a single NPU kernel.

4.2 End-to-End Results

Experimental Goal. The primary goal of our end-to-end evaluation is to quantify the trade-off between performance (latency) and correctness (fidelity) that developers face when using black-box vendor compilers. We aim to demonstrate that SHARD eliminates this trade-off by achieving the performance of an NPU-accelerated model with the numerical fidelity of a CPU-executed model. We specifically measure a) the wall-clock latency to encode a single image averaged over 100 runs and b) model fidelity by computing the average cosine similarity across layer outputs against an FP32 ground truth.

We compare SHARD against three baselines representing various trade-off points between latency and fidelity.

1. **RKNN-INT8:** This baseline represents the default toolchain. It applies black-box graph optimizations (e.g., fusion) and quantizes all tensor operations into INT8.
2. **RKNN-FP16:** This baseline represents a developer's attempt to fix fidelity issues by manually forcing FP16 precision. We decompose attention blocks into MatMul and Softmax to keep attention in FP16. The optimizer may still quantize and optimize the rest of the layer, such as the MLP.
3. **CPU-FP32:** This baseline measures pure PyTorch execution on the RK3588 CPU. This provides the ceiling for correctness and the floor for performance.

Table 1 summarizes the latency and fidelity of SHARD and the three baselines.

RKNN-INT8 achieves the lowest latency (1.40s) but produces mathematically useless output (0.02 Cosine Similarity). This occurs because the vendor compiler blindly applies symmetric INT8 quantization to the Transformer's activations. Since GELU outputs and Softmax probabilities in Vision Transformers exhibit extreme outliers (kurtosis > 100), standard quantization range-clipping destroys the signal. While the model runs efficiently on the NPU's INT8 units, the result is severely degraded with respect to the original model.

The CPU baseline delivers perfect fidelity but takes 30s per image, which is impractical for real-time applications. The heavy reliance on standard matrix multiplications saturates the CPU's NEON units, creating a latency bottleneck.

The RKNN-FP16 baseline attempts to strike a middle ground between latency and fidelity. Its fidelity reaches only 0.64 because unscaled FP16 values saturate the NPU's accumulators. Worse, its latency explodes to 19.63s (8.7× slower than SHARD). This latency regression is caused by the 1024×1024 attention transpose. Because it exceeds the NPU's 16K-element hardware limit, the runtime forces a fallback to the CPU. The cost of copying data back and forth between NPU SRAM and system DRAM dominates the execution time.

Meanwhile, SHARD achieves both low latency (2.24s) and high fidelity (0.95 cosine similarity). It is 13× faster than the CPU and 8.7× faster than the RKNN-FP16 baseline because it enforces **100% NPU residency**—our tiling primitives ensure no operator ever exceeds hardware limits, preventing CPU fallback. Simultaneously, it maintains high fidelity because it executes the entire model in FP16 while utilizing scaling to keep activation distributions within the numerical safe zone of the NPU's FP16 pipeline, preventing the saturation observed in the RKNN-FP16 baseline.

4.3 Attention Microbenchmark

To understand the efficacy of SHARD's primitives (Section 3), we perform a targeted microbenchmark on the Attention mechanism by executing a 1024x1024 attention operation. Attention is the primary bottleneck for Vision Transformers on edge NPUs. It combines large matrix multiplications with memory-intensive layout permutations (transposes) that often exceed the NPU's hardware capabilities.

As observed in Section 4.2, the RKNN-FP16 baseline was forced to execute the attention operation on the CPU. To better understand why, we initially observed that large fused kernels caused opaque compilation failures (throwing a REGTASK overflow 0xe010 error). To isolate the cause, we generated synthetic subgraphs with controlled increases in operand indices. Through empirical trial and error, we found that kernels with operand indices > 8191 consistently failed. This behavior revealed an undocumented 13-bit instruction register width limit on the NPU. Thus, vendor tools default to executing large transposes by falling back to the CPU.

Table 2. Attention microbenchmark performance results.

Configuration	CPU Fallback	Latency (mean ms)
RKNN-FP16	Yes	537.4
SHARD	No	167.7

Table 3. Cumulative fidelity ablation (Layer 0–11 sweep). Without scaling, fidelity collapses by Layer 5.

Layer	Cos Sim (SHARD)	Cos Sim (No Scaling)
L0	0.989	0.989
L1	0.978	0.760
L2	0.974	0.522
L5	0.968	0.112
L11	0.958	0.128

SHARD's barriers proactively fragment the graph to ensure no single kernel exceeds this addressable range. We evaluate the impact of SHARD's explicit loop unrolling and static tiling, which decomposes the 1024×1024 transpose into hardware-friendly 64×64 tiles. Table 2 demonstrates the results: preventing CPU fallback reduces attention latency by 3.2× (167.7ms vs 537.4ms) compared to the RKNN-FP16 baseline. Ideally, CPU fallback would only incur the cost of the operation itself; in practice, the overhead of context switching and moving trivial amounts of data between the NPU and CPU memory domains dominates execution time.

4.4 Quantization and Fidelity

Transformers, unlike ResNets, exhibit heavy-tailed activation distributions (e.g., in LayerNorm and Softmax inputs) that are highly sensitive to dynamic range clipping. As seen in Table 1, standard post-training quantization fails catastrophically for this model: naive INT8 quantization yields a cosine similarity of effectively zero (0.02). Even "weight-hardening" techniques like AWQ, which protect salient weights, fail because the error stems from *activation* outliers, not weight sensitivity. SHARD addresses this by enforcing a global scaling convention (Section 3.1.4). Without scaling, the FP16 values in the NPU pipeline accumulate noise that eventually saturates the mantissa limits.

To better understand this, Table 3 reports the cosine similarity of successive layers' output embeddings compared to CPU-FP32. We report results with SHARD's scaling technique and without. Without scaling, we observe a "saturation cliff" at Layer 5, where cosine similarity collapses ($0.98 \rightarrow 0.11$) due to FP16 mantissa saturation. By pre-scaling inputs by $\lambda = 0.1$ (compressing the dynamic range) and post-scaling by $\lambda = 10.0$, we keep activations within the sweet spot of the NPU's FP16 format, restoring end-to-end fidelity to > 0.95.

Profiling Effort. The compatibility profile for the RK3588 was constructed through a one-time manual profiling effort per hardware device. The 13-bit instruction register width limit was reverse-engineered by generating synthetic subgraphs with progressively increasing operand counts, requiring approximately 8 hours of iterative binary-search-style probing. The scaling factor λ was selected by sweeping a small candidate set $\{0.01, 0.1, 0.5, 1.0\}$ over our fidelity metric (cosine similarity), requiring one compile-and-validate cycle per candidate (approximately 2 hours total). Once discovered, these constraints are recorded in the compatibility profile and amortized across all models targeting the same platform. Automating this profile discovery via compiler-in-the-loop search is an explicit goal of future work (Section 6).

5 Related work

While open compiler frameworks like Apache TVM [4] and MLIR [16] provide robust, extensible infrastructure for deep learning compilation, they are often insufficient for edge NPUs like the RK3588. These frameworks rely on open-source drivers and detailed hardware specifications to generate efficient machine code. However, edge NPU vendors typically provide closed-source, proprietary toolchains (e.g., Rockchip's RKNN) and withhold the low-level ISA documentation required to build a custom backend. Consequently, practitioners are forced to use the vendor's black-box compiler, which is often brittle and optimized for traditional CNN architectures rather than modern Transformers. SHARD addresses this gap by acting as a pre-compiler. Instead of attempting to replace the vendor toolchain, SHARD rewrites the high-level graph (ONNX) into a form that the black-box compiler can successfully ingest and optimize, effectively steering it away from known failure modes (e.g., register overflows, SRAM exhaustion) without requiring access to the underlying ISA.

Kernel-level attention optimizations (e.g., FlashAttention [8]) provide algorithmic and kernel scheduling improvements on GPUs with open code generation. Our focus is deployability under opaque toolchains. Compiler toolchains such as MLIR [16], TVM [4], and XLA [32] support tiling and scheduling but assume a controllable backend. Graph optimizers (e.g., TASO [13]) optimize primarily for performance, as opposed to deployability on edge devices. ONNX Runtime Execution Providers (EPs) [20] offer a related graph-partitioning mechanism: subgraphs supported by a hardware EP (e.g., OpenVINO, TensorRT, RKNN) execute on the accelerator, while unsupported subgraphs fall back to the CPU. However, EP partitioning is best-effort and reactive—it accepts the graph as-is and partitions around unsupported operators rather than rewriting the graph to eliminate them. SHARD is complementary: by proactively rewriting the graph so that *all* operators are hardware-legal, SHARD ensures nothing remains for the CPU fallback path, achieving 100%

NPU residency where default EP partitioning cannot. Quantization techniques such as SmoothQuant [34], AWQ [17], and LLM.int8 [9] address outliers in other regimes; however, we show why standard quantization fails on small vision encoders due to spiky activations. Specifically, extreme outliers in the residual stream force a wide dynamic range that causes tiny visual signal updates to be rounded to zero in INT8, or overflow the hardware's FP16 variance accumulators.

6 Limitations and Future Work

The challenges encountered in our RK3588 proof-of-concept highlight the inherent difficulties of black-box optimization, motivating the future development of SHARD's automated vision:

- **Graph explosion:** Naive tiling can create too many nodes, triggering driver timeouts or artifact size limits (as discussed in Section 4). Our static tiling and sub-sharding strategies mitigate this by keeping artifacts under 10 MB.
- **Fusion reversion:** Downstream compilers may aggressively re-fuse split graphs or unsupported operators, leading to hardware faults like `REGTASK` overflows or fidelity collapse (Section 4). While our fusion barriers prevent this, the specific barrier patterns required can be highly backend-specific. This adversarial dynamic between precompilers and closed-source backends sets the stage for future research into dynamic, auto-discovered barriers.
- **Hardware profile maintenance:** Constraints and capabilities vary significantly across devices and SDK versions, requiring continuous updates to compatibility profiles [26, 29]. Our future vision replaces manual profile creation with automated, compiler-in-the-loop search (rewrite → compile → cache) to discover these limits programmatically.
- **Performance-optimality:** SHARD prioritizes deployability and fidelity over absolute peak performance. Achieving theoretical peak latency often requires kernel-level scheduling and open code generation (e.g., TVM [4] or FlashAttention [8]), which are unavailable in opaque edge toolchains.

7 Conclusion

Transformer deployment on edge accelerators is frequently limited by opaque, hard deployability constraints rather than raw compute. A deployability-first compatibility framework—built from a small set of reusable primitives—can convert transformer graphs into compiler-acceptable forms while preserving numerical fidelity.

While our current proof-of-concept relies on manually discovered heuristics for the RK3588, it successfully validates the broader vision of SHARD: a universal, profile-driven compatibility framework. As long as hardware vendors continue to prioritize proprietary, closed-source toolchains to protect their silicon IP, compatibility frameworks like SHARD will remain a necessary, distinct, and permanent layer in the systems machine learning deployment stack.

References

[1] AMD. 2026. AMD Introduces Ryzen AI Embedded Processor Portolio. https://www.amd.com/en/newsroom/press-releases/2026-1-5-amd-introduces-ryzen-ai-embedded-processor-portfol.html. Accessed: 2026-02-18.

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. doi:10.1145/3620665.3640366

[3] Ozan Baris, Yizhuo Chen, Gaofeng Dong, Liying Han, Tomoyoshi Kimura, Pengrui Quan, Ruijie Wang, Tianchen Wang, Tarek Abdelzaher, Mario Bergés, Paul Pu Liang, and Mani Srivastava. 2025. Foundation Models for CPS-IoT: Opportunities and Challenges. arXiv:2501.16368 [cs.LG] https://arxiv.org/abs/2501.16368

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[5] Coral. 2026. Coral Edge TPU. https://www.coral.ai/products/. Accessed: 2026-02-18.

[6] Coral. 2026. EdgeTPU Compiler. https://www.coral.ai/docs/edgetpu/compiler#system-requirements. Accessed: 2026-02-18.

[7] Coral. 2026. EdgeTPU Supported Operations. https://www.coral.ai/docs/edgetpu/models-intro#compatibility-overview. Accessed: 2026-02-18.

[8] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG] https://arxiv.org/abs/2205.14135

[9] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 2198, 15 pages.

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. arXiv:2010.11929 [cs.CV] https://arxiv.org/abs/2010.11929

[11] Google. 2026. Android Neural Networks API (NNAPI). https://developer.android.com/ndk/guides/neuralnetworks. Accessed: 2026-02-18.

[12] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861 [cs.CV] https://arxiv.org/abs/1704.04861

[13] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. doi:10.1145/3341301.3359630

[14] Kento Kawaharazuka, Jihoon Oh, Jun Yamada, Ingmar Posner, and Yuke Zhu. 2025. Vision-Language-Action Models for Robotics: A Review Towards Real-World Applications. *IEEE Access* 13 (2025), 162467–162504. doi:10.1109/access.2025.3609980

[15] Wasif Khan, Seowung Leem, Kyle B See, Joshua K Wong, Shaoting Zhang, and Ruogu Fang. 2026. A Comprehensive Survey of Foundation Models in Medicine. *IEEE reviews in biomedical engineering* 19 (2026), 283—304. doi:10.1109/rbme.2025.3531360

[16] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 2–14. doi:10.1109/CGO51591.2021.9370308

[17] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 87–100. https://proceedings.mlsys.org/paper_files/paper/2024/file/42a452cbafa9dd64e9ba4aa95cc1ef21-Paper-Conference.pdf

[18] LLVM. 2026. Torch MLIR. https://github.com/llvm/torch-mlir. Accessed: 2026-02-18.

[19] Andrés Marafioti, Orr Zohar, Miquel Farré, Merve Noyan, Elie Bakouch, Pedro Cuenca, Cyril Zakka, Loubna Ben Allal, Anton Lozhkov, Nouamane Tazi, Vaibhav Srivastav, Joshua Lochner, Hugo Larcher, Mathieu Morlon, Lewis Tunstall, Leandro von Werra, and Thomas Wolf. 2025. SmolVLM: Redefining small and efficient multimodal models. arXiv:2504.05299 [cs.AI] https://arxiv.org/abs/2504.05299

[20] Microsoft. 2026. ONNX Runtime. https://onnxruntime.ai/. Accessed 2026-02-13.

[21] NVIDIA. 2026. The Ultimate Platform for Physical AI and Robotics. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/. Accessed: 2026-02-18.

[22] IREE Org. 2026. IREE: Intermediate Representation Execution Environment. https://iree.dev/. Accessed: 2026-02-18.

[23] Qualcomm. 2025. Qualcomm Snapdragon Neural Processing Engine (SNPE). https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk. Accessed: 2026-02-18.

[24] Qualcomm. 2026. Hexagon SDK. https://www.qualcomm.com/developer/software/hexagon-npu-sdk. Accessed: 2026-02-18.

[25] Qualcomm. 2026. Qualcomm Hexagon. https://www.qualcomm.com/processors/hexagon. Accessed: 2026-02-18.

[26] Qualcomm. 2026. Supported ONNX Ops. https://docs.qualcomm.com/doc/80-63442-10/topic/supported_onnx_ops.html. Accessed 2026-02-13.

[27] Rockchip. 2024. RK3588 NPU SRAM usage. https://github.com/rockchip-linux/rknpu2/blob/master/doc/RK3588_NPU_SRAM_usage.md. Accessed 2026-02-13.

[28] Rockchip. 2025. Rockchip RKNN Toolkit 2. https://github.com/rockchip-linux/rknn-toolkit2. Accessed: 2026-02-18.

[29] Rockchip. 2026. RKNN Supported Operators. https://github.com/rockchip-linux/rknpu2/blob/master/doc/RKNN_Compiler_Support_Operator_List_v1.5.2.pdf. Accessed: 2026-02-18.

[30] Rockchip. 2026. Rockchip. https://www.rock-chips.com/a/en/index.html. Accessed: 2026-02-18.

[31] Ltd. Shenzhen Xunlong Software Co. 2026. OrangePi 5 Max. http://www.orangepi.org/html/hardWare/computerAndMicrocontrollers/details/Orange-Pi-5-Max.html. Accessed: 2026-04-09.

[32] TensorFlow. 2026. XLA: Accelerated Linear Algebra. https://www.tensorflow.org/xla. Accessed: 2026-02-18.

[33] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3315508.3329973

[34] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1585, 13 pages.

[35] Xiaohua Zhai, Basil Mustafa, Alexander Kolesnikov, and Lucas Beyer. 2023. Sigmoid Loss for Language Image Pre-Training. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*. 11941–11952. doi:10.1109/ICCV51070.2023.01100