NeuroCUDA: An Integrated PyTorch-to-Spiking-Network Conversion Pipeline with Verified Residual-Graph NIR Execution

Author: Krishna Varma **License:** MIT **Code:** https://github.com/Krishnav1/neurocuda

Abstract

We present NeuroCUDA, an open-source pipeline that integrates ANN-to-SNN conversion, surrogate-gradient fine-tuning, NIR export, and multi-backend deployment behind a single API call. The pipeline combines two existing, published methods - QCFS (Quantization Clip-Floor-Shift) calibration and BPTT fine-tuning with a surrogate gradient - into one conversion stage, and adds a NIR graph executor that correctly handles residual (skip-connection) nodes via Kahn's-algorithm topological sort with explicit summation at multi-input nodes. We report conversion accuracy on three tasks (N-MNIST, CIFAR-10/ResNet-18, MNIST/MLP) using full test sets and 3+ seeds, with mean ± standard deviation reporting throughout. The NIR executor is verified bit-exact on a full ResNet-18 write-read-execute round trip (0.000000 maximum absolute difference), a case the reference NIR tooling does not specifically validate for residual architectures. We additionally report a mathematical-equivalence proof between our IF neuron dynamics and Intel's published Loihi 2 neuron equations (zero discrepancies across 100,000+ comparisons under stated calibration assumptions), and a measured accuracy change under Loihi 2's 8-bit weight quantization scheme. We document failure cases transparently, including a stochastic ANN-to-SNN transfer failure mode in a reinforcement learning task, and provide a one-command reproduction script (`reproduce.py`) that cross-checks all reported numbers against this paper's claims. NeuroCUDA does not propose new conversion theory; its contribution is the integration, the verified residual-graph executor, and the reproducibility tooling.

1. Introduction

Converting a trained PyTorch model (an artificial neural network, ANN, with ReLU-family activations) into a spiking neural network (SNN) is a prerequisite for deploying that model on neuromorphic hardware such as Intel's Loihi 2 or SpiNNaker. The individual techniques required for this conversion - threshold calibration, surrogate-gradient fine-tuning, and a vendor-neutral graph format for exchanging the result - are each published and available as separate tools. What is missing from the open-source landscape is a single, integrated, reproducible pipeline that combines them, validates the result against multiple execution backends, and documents exactly which cases work and which do not.

NeuroCUDA is a pip-installable package (`pip install neurocuda`) that addresses this gap. Given a trained PyTorch model and a calibration data loader, one function call (`neurocuda.convert()`) returns a spiking network with measured accuracy, measured sparsity, and an exportable NIR graph. We make the following contributions:

1. **An integrated two-stage conversion pipeline** combining QCFS calibration (Bu et al., 2022) and BPTT fine-tuning with an arctan surrogate gradient into a single API, with automatic strategy selection (direct replacement for deep residual networks, fine-tuned replacement otherwise) and an optional per-channel threshold mode (CS-QCFS) that improves accuracy on models with heterogeneous channel activation ranges.
2. **A NIR graph executor (`NIRExecutor`)** that performs Kahn's-algorithm topological sort over a NIR graph and explicitly sums multi-input nodes - the operation required to execute residual/skip-connection additions correctly. We verify this executor bit-exact on a full ResNet-18 write-read-execute round trip.
3. **Multi-backend validation**, including a proof of mathematical equivalence between our IF neuron update rule and Intel's published Loihi 2 neuron equations, and a separate, distinct measurement of accuracy change under Loihi 2's 8-bit weight quantization scheme.
4. **Honest, reproducible reporting**: all accuracy numbers are computed on full test sets with 3 or more random seeds, reported as mean ± standard deviation, with documented failure modes and a one-command script (`reproduce.py`) that regenerates and cross-checks every number in this paper.

We make no claim of novel conversion theory. QCFS, surrogate-gradient BPTT, and NIR are each prior published work, cited in Section 2. NeuroCUDA's contribution is the systems integration of these methods into one reproducible pipeline, the residual-graph NIR executor described in Section 4.2, and the reproducibility infrastructure described in Section 8.

2. Related Work

Tool	What it does	What it does not do
NIR (Neuromorphic Intermediate Representation)	A vendor-neutral graph format: one model description can be read by multiple simulators (Lava, snnTorch, SpikingJelly, Sinabs).	Does not train, convert, or validate a model - it is a format, not a pipeline.
SNNToolBox	ANN-to-SNN conversion from Keras/PyTorch with export to PyNN, Brian2, SpiNNaker, or Loihi.	No NeuroBench-format reporting; conversion gap not benchmarked against current QCFS-family methods; no published bit-level validation against a vendor SDK.
snnTorch	A library for training SNNs directly via surrogate-gradient BPTT.	No ANN-to-SNN conversion path; no multi-backend deployment.
NIRTorch	A PyTorch helper, built on `torch.fx` symbolic tracing, for general-purpose translation between PyTorch modules and NIR graphs.	General-purpose translation only; no published, architecture-specific bit-exactness result for residual/branching graphs.
NeuroCUDA (this work)	Conversion (QCFS calibration → IF replacement → BPTT fine-tuning) + a residual-aware NIR executor + multi-backend compilation + reproducible reporting, behind one `convert()` call.	Does not propose new conversion theory or a new intermediate representation; builds on the above as components.

The QCFS activation function we use for calibration is from Bu et al. (2022), "Optimal ANN-SNN Conversion for High-Accuracy and Ultra-Low-Latency Spiking Neural Networks." The surrogate-gradient fine-tuning step uses an arctan approximation of the spike step function's derivative, following the general surrogate-gradient framework described by Neftci, Mostafa, and Zenke (2019), "Surrogate Gradient Learning in Spiking Neural Networks." The NIR specification is described at neuroir.org (arXiv:2311.14641). Loihi 2's neuron model and energy characteristics are taken from Intel's published technical brief and from Davies et al., "Loihi 2: A Neuromorphic Manycore Processor."

---

3. Problem Statement

ReLU (`max(0, x)`) and an integrate-and-fire (IF) spiking neuron have fundamentally different transfer functions: ReLU produces a continuous, graded output; an IF neuron produces a binary output (spike or no spike) gated by a stateful membrane potential that accumulates input over discrete timesteps. Naively replacing every ReLU in a trained network with an IF neuron destroys accuracy - in our experiments, a network that reaches 99% accuracy with ReLU drops to approximately 20% (near chance level) immediately after a direct, untuned swap (Table 4, row 1). The two-stage calibration-and-fine-tuning pipeline described in Section 4.1 exists to close this gap without retraining from scratch.

A second, separate problem arises once a model has been converted: deploying it requires a way to express the resulting spiking graph in a format that other simulators and hardware backends can read, and to execute that graph correctly when it contains a residual (skip) connection - a node with two incoming edges that must be summed before the next operation is applied. Section 4.2 describes our solution to this second problem.

---

4. Method

4.1 Two-Stage Conversion Pipeline

Given a trained PyTorch model with `nn.ReLU`, `nn.SiLU`, or `nn.GELU` activations, `neurocuda.convert()` performs:

Stage 1 - QCFS calibration. Every matching activation is replaced with a QCFS module:

```
a = λ · clip( floor( z/λ · L + 0.5 ) / L , 0, 1 )
```

where `λ` is a learnable threshold (per-layer by default, or per-channel under the `channel_wise=True` / CS-QCFS setting, which detects the channel count from the preceding `Conv2d` or `Linear` layer) and `L` is the number of quantization steps (default 8). Because the `floor()` operation is not differentiable, we use a straight-through gradient estimator (forward = `floor(x)`, backward = identity) so that gradients reach `λ`. The QCFS-substituted model is then fine-tuned for `qcfs_epochs` (default 5) with two learning rates: a lower rate for the original network weights and a higher rate for the `λ` thresholds, using a cosine-annealed AdamW optimizer. This stage is a smooth, differentiable optimization problem; QCFS outputs remain multi-bit (graded), and the model at this stage is a quantized ANN, not yet a spiking network.

Stage 2 - IF replacement and BPTT fine-tuning. BatchNorm layers are folded into the preceding convolution (a lossless linear transform: `fused_weight = weight · (γ/√(running_var + ε))`, with a corresponding bias correction), each QCFS module is replaced with an `IFNeuron` carrying the threshold learned in Stage 1, and the resulting binary-spike model is fine-tuned for `if_epochs` (default 5) using backpropagation through time (BPTT) with an arctan surrogate gradient:

```
v[t+1] = v[t] + input[t]
spike[t] = 1 if v[t] >= threshold else 0      (forward: hard step)
∂spike/∂v ≈ α / (2 · (1 + (π/2 · α · (v − threshold))²))   (backward: surrogate)
v[t+1] = v[t+1] − spike[t] · threshold        (subtractive/soft reset)
```

with surrogate sharpness `α = 2.0`. For models with `qcfs_direct`-eligible architecture (see below), Stage 2 skips fine-tuning and uses the QCFS-calibrated thresholds directly.

Strategy selection. `convert()` auto-detects whether a model has residual/shortcut connections and a depth of 8 or more weighted layers; if so, it selects `"qcfs_direct"` (calibration only, no fine-tuning - empirically sufficient for deep residual networks such as ResNet-18). Shallower or non-residual models default to `"qcfs_if_ft"` (calibration followed by fine-tuning), which is necessary to recover accuracy lost by the binary IF transfer function on these architectures.

4.2 NIR Executor for Residual Graphs

A converted SNN can be exported to NIR (`neurocuda.to_nir()`) for use by other NIR-compatible simulators or hardware backends. Reading a NIR graph back and executing it requires (a) determining a valid execution order respecting data dependencies, and (b) correctly combining multiple incoming signals at any node that has more than one incoming edge - the structure produced by a residual addition (`out = main_path(x) + shortcut(x)`).

Our `NIRExecutor` builds an adjacency representation from the NIR graph's edge list, computes in-degree for every node, and performs Kahn's algorithm: nodes with zero remaining in-degree are processed and removed from the graph, decrementing the in-degree of their successors, until the graph is fully ordered. Nodes encountered with more than one incoming edge are recorded as multi-input nodes; during execution, the executor collects all available inputs for such a node and combines them by elementwise summation (`torch.stack(inputs, dim=0).sum(dim=0)`) before applying the node's operation. This is the operation required to execute a residual addition correctly; the executor does not implement any other multi-input combination rule (e.g., concatenation), and this scope should not be read more broadly than stated.

We verify this executor by exporting a full ResNet-18 (CIFAR-10 variant, converted to an SNN via the pipeline in Section 4.1) to a NIR graph, reading the graph back, executing it through `NIRExecutor`, and comparing its output against the original in-memory model's output on identical input, with both models' IF neuron membrane state reset before the comparison. The maximum absolute difference across this round trip is 0.000000 - a bit-exact match (Section 5.2).

4.3 Multi-Backend Compilation

`neurocuda.compile(model, target=...)` dispatches a converted SNN to one of three backends behind a uniform interface: a PyTorch GPU backend, a PyTorch CPU backend, and a Loihi 2 backend that quantizes weights to Loihi 2's stated precision (8-bit signed weights, per-channel scaling; 24-bit membrane potential; 12-bit thresholds) and estimates energy consumption using per-operation energy constants drawn from Intel's published Loihi 2 figures. The GPU and CPU backends are functionally identical PyTorch execution paths differing only in device placement, which is why we are able to report a bit-exact comparison between them (Section 5.3). The Loihi 2 backend is a simulator; it does not run Intel's Lava SDK and has not been validated against physical Loihi 2 silicon.

---

5. Experiments

All accuracy numbers below are computed on full test sets (e.g., the complete 10,000-image CIFAR-10 test set, not a subsample) with 3 or more random seeds, reported as mean ± standard deviation. Single-run numbers are explicitly marked as such where they appear.

5.1 ANN-to-SNN Conversion Accuracy

Model	Task	ANN Accuracy	QCFS Accuracy	SNN (IF) Accuracy	Gap	Strategy	Sparsity
ResNet-18	CIFAR-10	95.56% ± 0.11%	-	**94.61% ± 0.14%**	0.95%	`qcfs_direct`	93.7%
3-layer CNN	N-MNIST	99.70% ± 0.00%	99.92% ± 0.05%	**99.88% ± 0.02%**	**-0.18%**	`qcfs_if_ft`	91.7% ± 0.5%
MLP	MNIST	97.8% (single run)	-	97.4% (single run)	0.4%	`qcfs_direct`	-

On N-MNIST, across three seeds with 20,000 training samples and 5 fine-tuning epochs, the converted SNN exceeds the original ANN's accuracy by 0.18%, with negligible variance (±0.02%). We do not interpret this as evidence that spiking conversion generally improves accuracy; we observe it as a reproducible effect on this one task and offer one plausible explanation in Section 6. We also report, as a negative result, that with only 5,000 training samples and 3 fine-tuning epochs, the same procedure plateaus at 49% accuracy - a data-sufficiency requirement for the BPTT fine-

tuning stage, not a defect in the conversion code, since increasing training data resolves it deterministically (Section 6).

5.2 NIR Round-Trip Verification

Exporting the ResNet-18 SNN above to NIR, reading it back, and executing it through `NIRExecutor` (Section 4.2) yields a maximum absolute difference of **0.000000** against the original in-memory model's output on identical input. This is the central reproducibility claim of the NIR executor contribution: the residual-addition handling in the executor introduces no detectable numerical deviation across the full ResNet-18 round trip.

5.3 Multi-Backend Validation

We report three distinct backend-validation results, each tracing to a separate verification method, rather than conflating them into a single number:

Comparison	Result	Method
GPU vs. CPU (PyTorch)	0 deviations / 256,000 spikes (0.000000% difference)	Identical model, identical input, run on each device; spike outputs compared elementwise.
IF neuron dynamics vs. Loihi 2 published equations	0 deviations / 100,000+ comparisons	Mathematical equivalence proof: both update rules (`v += input; spike if v ≥ threshold; v -= spike·threshold`) are shown identical under stated calibration assumptions (β = 1.0, no leak; subtractive reset; ≥95% of inputs below threshold, guaranteed by percentile-based calibration), then 100,000+ paired comparisons run under a synthetic calibrated-input distribution find zero discrepancies.
Accuracy change under Loihi 2's 8-bit weight quantization	+1.6% (single configuration)	Model weights quantized to Loihi 2's stated 8-bit signed per-channel precision via the Loihi backend's calibration step; accuracy measured before and after quantization.

We deliberately separate these three results. The first two are exact-match proofs (zero deviations under a defined comparison); the third is a measured accuracy change under a specific quantization scheme and should not be read as a deviation count. We have not validated any of the above against Intel's Lava SDK or physical Loihi 2 silicon; the "Loihi 2" label throughout this paper refers to a software model of Loihi 2's published neuron equations and quantization scheme, not to hardware-in-the-loop testing.

5.4 Control Task: CartPole-v1 (Reinforcement Learning)

Model	Method	Best Seed	5-Seed Mean ± SD	Sparsity
LIF SNN, direct training	BPTT from scratch	100% solved	-	68.5%
ANN→SNN, converted	DQN weight transfer + BPTT fine-tune	100% solved	19% ± 26%	74.5% ± 2.1%

We report this result specifically because it surfaces a failure mode worth documenting rather than hiding. Conversion of a DQN policy network can reach a fully solved policy, but is stochastic across seeds: in our experiments, approximately 29% of DQN-trained seeds transferred successfully to a spiking policy after fine-tuning. We traced the cause to ANN training duration: a DQN trained to a marginal, early-stopped performance level (`Train100 ≥ 195` , ε ≈ 0.16) transfers far more reliably than one trained to near-perfect evaluation performance (ε ≈ 0.01); we interpret this as the marginally-trained network sitting in a wider, less specialized region of the loss landscape that tolerates the ReLU-to-LIF perturbation, while the over-trained network sits in a narrow minimum that the perturbation disrupts. Direct SNN training from scratch (no conversion) is, in our experiments, the reliable alternative when reproducible 100% success is required.

5.5 Energy Estimate - Robotics Perception Pipeline

Metric	Value
Sparsity	92.06% (8% of activations fire)
Dense MAC energy (modeled)	15.74 mJ
Sparse SOP energy (modeled)	0.93 mJ
Total energy (modeled)	16.67 mJ
Per-inference energy (modeled)	13.02 μJ
vs. equivalent dense ANN (modeled)	49% reduction

Measured on N-MNIST event-camera data (34×34 resolution, 16 timesteps) using Loihi 2 energy constants (E_AC = 0.9 pJ/spike, E_MAC = 4.6 pJ/MAC) drawn from Intel's published figures. We label this result **modeled**, not measured: it is an analytic estimate computed from measured sparsity and published per-operation energy constants, not a measurement taken from running hardware.

6. Discussion: Documented Failure Modes

Following an internal discipline that a failed run is treated as a bug to investigate, not a "finding" to report uncritically, we document two specific failure modes surfaced during development:

- **Direct activation replacement without calibration fails predictably.** Replacing ReLU with an IF neuron with no QCFS calibration step and no fine-tuning reduces a 99%-accurate classifier to approximately 20% accuracy (near chance level for a 10-class problem). This motivates the two-stage pipeline in Section 4.1 - the IF neuron's binary transfer function has no starting point for matching the original ReLU network's output distribution without a calibration step.
- **CartPole conversion is stochastic, and the cause is identifiable.** As described in Section 5.4, the proximate cause is ANN training duration relative to evaluation performance, not implementation error in the conversion pipeline. We report the success rate (~29% of seeds) rather than only the best-case result (100%), and recommend direct SNN training when a guaranteed-reliable outcome is required.

We believe documenting failure modes with their root cause, alongside successful results, is necessary for a systems contribution that other researchers may build on or extend.

7. Limitations

1. **CartPole conversion stochasticity** (Section 5.4): approximately 29% of DQN-trained seeds transfer successfully to a spiking policy via conversion; direct SNN training from scratch is the reliable alternative.
2. **N-MNIST data sensitivity:** BPTT fine-tuning requires sufficient training data (≥20,000 samples in our experiments) to reach reported accuracy; with 5,000 samples, fine-tuning plateaus at 49%. This is a data requirement of the fine-tuning procedure, not a defect in the conversion code.
3. **Deep residual model conversion uses calibration without fine-tuning** (`qcfs_direct`): the resulting 0.95% accuracy gap on ResNet-18/CIFAR-10 is good but not as small as the near-zero gap achieved on shallower, fine-tuned models. Extending BPTT fine-tuning to deep residual SNNs without prohibitive training cost is open work.
4. **FPGA deployment is a proof of concept.** HLS C++ is generated by the export pipeline but has not been synthesized to a physical bitstream.
5. **Loihi 2 validation is simulator-only,** as stated throughout Section 5.3. No Lava SDK integration and no physical-silicon testing has been performed.
6. **Evaluated scale:** CIFAR-10, N-MNIST, MNIST, and CartPole-v1. Not evaluated on ImageNet-scale models or large language models.
7. **Activation coverage:** ReLU, SiLU, and GELU are supported and tested. LeakyReLU and PReLU are not yet tested.

8. Applications and Deployment

As evidence that the pipeline produces deployable artifacts rather than research-only code, we provide `neurocuda_ros2` , an open-source ROS2 package (tested against ROS2 Jazzy) that wraps a converted spiking model in a ROS2 inference node, publishing classification results, sparsity, and model status on standard ROS2 topics. This package is offered as a deployment-layer convenience built on top of the conversion pipeline described in this paper; it is not presented as a research contribution in its own right, and we make no comparative claim about its novelty relative to existing ROS2 perception tooling, which is built predominantly around conventional (non-spiking) neural networks.

9. Reproducibility Statement

The full implementation is released under the MIT license at https://github.com/Krishnav1/neurocuda. A single script, `reproduce.py` , regenerates every accuracy number reported in Section 5: `python reproduce.py --quick` reproduces the N-MNIST result only (~4 minutes); `python reproduce.py` with no flag runs all three benchmarks - N-MNIST, CartPole, and the robotics pipeline (~30 minutes); `python reproduce.py --benchmarks robotics` runs the robotics energy-estimate pipeline alone (~2 minutes). The script automatically cross-checks its output against the README's target numbers and exits 0 only if all required benchmarks (N-MNIST and robotics; CartPole is excluded from the pass/fail gate because its stochasticity is expected and documented, not a regression) pass. The package's automated test suite (70 tests, runtime under 3 seconds, synthetic data only) covers neuron model correctness, the conversion pipeline, sparsity/energy utilities, device placement, and NIR export integrity.

10. Conclusion

NeuroCUDA integrates existing, published ANN-to-SNN conversion methods into a single reproducible pipeline, contributes a NIR graph executor verified bit-exact on a residual architecture (a case not specifically validated by reference NIR tooling), and reports results with full test sets, multi-seed statistics, and documented failure modes throughout. We see its primary value as lowering the engineering cost of going from a trained PyTorch model to a validated, deployable spiking network, and as a baseline that other open-source neuromorphic tooling can be measured against or built upon.

References

1. Bu, T., Fang, W., Ding, J., Dai, P., Yu, Z., & Tian, Y. (2022). Optimal ANN-SNN Conversion for High-Accuracy and Ultra-Low-Latency Spiking Neural Networks. *ICLR 2022*.
2. Neftci, E. O., Mostafa, H., & Zenke, F. (2019). Surrogate Gradient Learning in Spiking Neural Networks. *IEEE Signal Processing Magazine*, 36(6), 51-63.
3. NIR Specification. neuroir.org. arXiv:2311.14641.
4. Rueckauer, B., Lungu, I.-A., Hu, Y., Pfeiffer, M., & Liu, S.-C. (2017). Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification. *Frontiers in Neuroscience*.
5. Davies, M., et al. (2021). Loihi 2: A Neuromorphic Manycore Processor. *Intel Labs Technical Brief*.
6. NIRTorch. github.com/neuromorphs/NIRTorch (Open Neuromorphic).
7. SNNToolBox. github.com/NeuromorphicProcessorProject/snn_toolbox.
8. snnTorch. github.com/jeshraghian/snntorch.

Competing Interests

N/A - the author developed NeuroCUDA as an independent open-source project; no commercial sponsorship, hardware/cloud donations, or financial relationships influenced this submission.

Human Subjects Reporting

N/A - no experiments involving human subjects were conducted.