December 2025 | Volume 1 | Issue 3 | Article No. 25212

Journal of Smart Sensors and Computing

Research Article | *Open Access* |

The Unified Neuromorphic Assembly Layer for Hardware-Agnostic Compilation in Neuromorphic Computing

Ganesh D. Jadhav,[1,*] Rahul V. Dagade,[2] Sushant Jakhade,[1] Kshitij Jadhav,[1] Rutu Hinge[1] and Swarada Joshi[1]

[1] *Department of Information Technology, Vishwakarma Institute of Technology, Pune, Maharashtra, 411037, India*
[2] *Department of Computer Engineering, Vishwakarma Institute of Technology, Pune, Maharashtra, 411037, India*
**Email:* jadhavganesh874@gmail.com (G. Jadhav)

Abstract

The programming of neuromorphic assembly has advanced steadily, providing essential tools and paradigms to help connect the gap between abstract Spiking Neural Network (SNN) models and brain-inspired computing hardware. This work presents UNAL (Unified Adaptive, Hardware-Agnostic Neuromorphic Assembly Layer). This compilation framework translates high-level Spiking Neural Network (SNN) models into portable, spike-level assembly across heterogeneous neuromorphic platforms. UNAL introduces a unified intermediate representation (UNAL-IR), a compact instruction set, and an optimization-driven mapping pipeline that jointly addresses latency, energy efficiency, routing congestion, and adaptability. Quantitative evaluation on standard SNN benchmarks (DVS Gesture and CIFAR-10 SNN) mapped to Intel Loihi 2 demonstrates 18–32% latency reduction, 21–38% energy savings, and 25–40% lower routing congestion compared to Loihi-native and platform-specific tool chains. A smart-city surveillance case study further validates the deployment of real-time edge computing. These results establish UNAL as a scalable and future-ready neuromorphic compiler infrastructure.

Keywords: Spiking neural networks; Neuromorphic compiler; Hardware-agnostic SNN; Intermediate representation; Instruction set.

Received: 14 November 2025; Revised: 20 December 2025; Accepted: 22 December 2025; Published Online: 23 December 2025.

1. Introduction

Neuromorphic computing has emerged as a promising alternative to conventional von Neumann architectures because it closely mirrors the event-driven signaling and massive parallelism observed in biological neural systems. By integrating memory and computation, neuromorphic systems alleviate the memory–processing bottleneck inherent in traditional architectures. As modern applications increasingly demand ultra-low-latency inference and energy-efficient intelligence at the edge, spiking neural networks (SNNs) and neuromorphic processors have attracted significant attention.[1,2] These systems operate using sparse, asynchronous spike-based communication, enabling computation that is fundamentally different from conventional deep learning models.

Despite these advantages, translating high-level SNN models into executable low-level instructions compatible with specific neuromorphic hardware platforms remains a major challenge. This process, often referred to as neuromorphic assembly programming, requires careful handling of spike routing, hardware resource allocation, memory organization, on-chip timing constraints, and parallel execution patterns.[3,4] The challenge is exacerbated by the growing diversity of neuromorphic architectures, which differ in crossbar designs, communication fabrics, spike encoding schemes, and on-chip learning support, thereby limiting model portability across platforms.[5,6]

Early efforts in neuromorphic compilation primarily

DOI: https://doi.org/10.64189/ssc.25212

focused on direct mappings tailored to specific hardware platforms. For example, Rueckauer *et al*.[7] introduced NxTF, an API–compiler stack designed to efficiently deploy deep spiking neural networks (SNNs) on Intel's Loihi processor by optimizing spike traffic and crossbar utilization. However, this approach is inherently tied to the Loihi architecture and does not generalize to other neuromorphic platforms.[7] Song *et al*.[8] later proposed a dataflow-driven compilation framework based on cluster partitioning and self-timed scheduling to support asynchronous, event-driven execution, which subsequently evolved into DFSynthesizer-a method that converts spike activity into hardware-executable dataflow instructions.

Complementary research has explored improved spatial and temporal mapping strategies. SpiNeMap clusters synapses based on functional proximity to reduce routing overhead and latency,[9] while NeuMap extends this approach using greedy heuristics combined with meta-optimization techniques to enhance SNN placement across heterogeneous neuromorphic cores.[10] Additional studies have investigated modular design patterns, reusable building blocks, and FPGA-based spike-coding mechanisms to support flexible neuromorphic prototyping.[11,12] Collectively, these efforts establish a growing foundation for neuromorphic compilers and assembly-level SNN deployment.

Despite these advancements, several limitations remain. Most existing frameworks are designed for a single hardware target, limiting interoperability as new neuromorphic processors emerge. Furthermore, many toolchains lack native support for adaptive dynamics, online learning, and synaptic plasticity-capabilities essential for next-generation neuromorphic intelligence. Current compilers also provide only partial automation, requiring manual intervention to resolve spike routing, concurrency, and resource contention issues.

To overcome these challenges, this work proposes a hardware-agnostic neuromorphic compilation framework that automatically synthesizes spike-level instructions while supporting dynamic behaviors such as adaptation and plasticity. The proposed approach introduces a unified abstraction layer over heterogeneous architectures, enabling consistent SNN deployment across diverse hardware platforms. In addition, it integrates optimization strategies for routing, event scheduling, and workload partitioning, thereby improving performance portability and reducing developer effort.

The primary contribution of this work is the extension of neuromorphic assembly programming beyond hardware-specific solutions through an extensible and generalizable compilation pipeline. This contribution is significant as the neuromorphic ecosystem continues to diversify with emerging chips, reconfigurable fabrics, and hybrid analog–digital substrates. A hardware-agnostic, automated, and adaptive compilation strategy is therefore essential to fully realize the potential of neuromorphic computing in real-world intelligent systems.

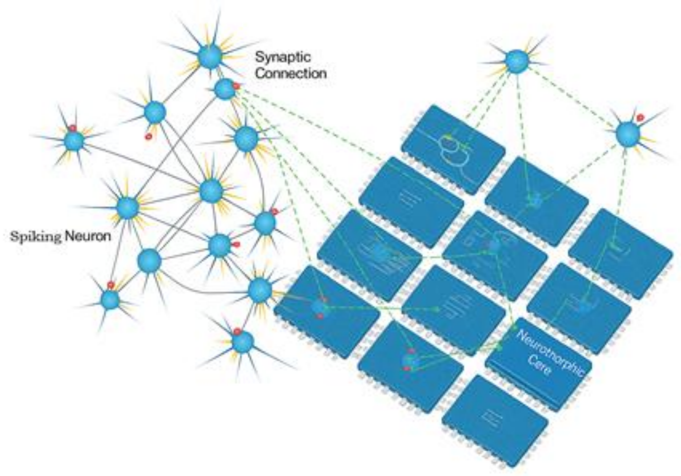


Fig. 1: Spiking neural network mapped onto neuromorphic computing cores.

2. Literature review

2.1 Overview: Why compiler and mapping research matters

Neuromorphic systems operate under stringent, hardware-specific constraints, including limited crossbar sizes, specialized spike-routing fabrics, restricted tile-to-tile bandwidth, and tightly coupled on-chip memory. These constraints make the automatic translation of high-level spiking neural network (SNN) models into efficient, executable low-level instructions a central bottleneck for adoption. Recent benchmark studies and surveys emphasize both the rapid growth of neuromorphic hardware platforms and the persistent fragmentation of software toolchains, highlighting limited interoperability across systems.[1,6] Consequently, compiler and mapping research has emerged as a critical enabler for scalable and deployable neuromorphic computing.

2.2 Hardware-specific compilers and toolchains

Early research efforts focused on platform-specific compilers that maximize performance on individual neuromorphic chips. NxTF exemplifies this approach by providing a Keras-like interface and compiler stack tailored to Intel's Loihi architecture. By exploiting Loihi's neuron models, routing infrastructure, and weight-sharing mechanisms, NxTF achieves high utilization and competitive accuracy on its target hardware.[7,4] While such hardware-specific compilers deliver strong results, they embed architectural assumptions-such as fixed crossbar dimensions, routing topologies, and neuron primitives-deep within their intermediate representations and optimization passes.[5] This tight coupling results in brittle toolchains that require significant redesign to support emerging neuromorphic substrates.

2.3 Dataflow and cluster-based mapping frameworks

To improve portability and analytical tractability, several frameworks adopt dataflow-driven or cluster-based compilation strategies. DFSynthesizer introduces a pipeline that partitions SNN workloads into hardware-constrained clusters, models them as synchronous dataflow graphs, and schedules execution under bandwidth and latency constraints.[8] Complementary approaches such as SpiNeMap emphasize synaptic clustering based on functional proximity to reduce routing overhead and inter-core communication.[9] Although these methods improve mapping quality and enable performance analysis, they often rely on simplified abstractions such as homogeneous tiles or crossbars. As a result, they struggle to capture the diversity of modern neuromorphic fabrics, including irregular interconnects and mixed analog–digital architectures.[4,13]

2.4 Spatial/temporal placement & optimization heuristics

Placement-oriented tools such as NeuMap apply fast heuristic and metaheuristic strategies to reduce network-on-chip latency and energy consumption by colocating neurons with high communication affinity.[10] These techniques are often combined with communication-aware partitioning to reduce spike traffic in large-scale deployments.[13] While heuristic placement scales well, it typically optimizes a narrow objective, such as communication locality. Important aspects—including timing jitter, asynchronous execution behavior, and online plasticity—are often excluded, limiting robustness in heterogeneous and adaptive neuromorphic systems.[14]

2.5 Modular architectures and reusable design patterns

Several studies propose modular and parameterized spike-based building blocks that encapsulate common computations such as feature extraction, pooling, and normalization.[11] These abstractions simplify SNN composition and improve reasoning about verification and resource usage.

Despite these advantages, modular patterns are not yet first-class citizens in neuromorphic compilers. Without a shared abstraction layer or pattern-aware intermediate representation, such constructs remain library-level conveniences rather than integrated compilation primitives.[15]

2.6 FPGA implementations and prototyping platforms

FPGA-based SNN implementations provide flexible platforms for algorithm–hardware co-design and support custom spike encodings and online learning mechanisms.[12,16] These platforms enable rapid experimentation prior to committing designs to silicon. However, FPGA architectures differ significantly from neuromorphic chips in communication models and timing semantics. Consequently, mapping strategies effective on FPGAs do not directly translate to specialized neuromorphic fabrics, and many FPGA-based efforts emphasize functional validation rather than full compilation automation.[16]

2.7 Surveys, trends, and device-level progress (materials to chips)

Comprehensive surveys and benchmark studies document strong progress in neuromorphic computing, spanning large-scale digital processors, mixed-signal systems, and emerging synaptic devices such as ferroelectric FETs.[1,6,17] These works consistently emphasize the need for cross-layer toolchains that jointly account for device characteristics, learning rules, and system-level constraints.

As the hardware landscape continues to diversify, existing toolchains—often built around rigid architectural abstractions—struggle to remain broadly applicable, widening the gap between device innovation and software support.[2,3]

2.8 Comparative analysis & limitations across approaches

Generality vs. performance trade-off: Platform-specific compilers (e.g., NxTF) achieve strong performance on a single chip but do not generalize; dataflow-based and clustering frameworks are more portable conceptually but still assume specific hardware abstractions (tiles, crossbars).[7]

Support for adaptation & online learning: Few toolchains include built-in, first-class support for plasticity, continuous adaptation, or online learning; most focus on static mapping of inference workloads.[14]

Automation depth: Several systems require manual tuning (e.g., cluster sizes, mapping parameters, scheduling tweaks). End-to-end automation that jointly optimizes accuracy, latency, energy, and hardware constraints remains limited.[9, 10]

Hardware heterogeneity: The expanding variety of synaptic device technologies and interconnect topologies makes a one-size-fits-all compiler difficult; current tools either abstract away critical device details or hard-wire assumptions that break portability.[11,15]

2.9 Identified research gap motivating the UNAL framework

Lack of hardware-agnostic compilation layers that can expose a consistent intermediate representation while capturing significant device-specific constraints (timing jitter, analog non-idealities, interconnect differences). Existing compilers either over-specialize (high performance, low portability) or over-abstract (portability at the cost of fidelity).

Insufficient native support for adaptive dynamics and online learning within compilation pipelines; existing toolchains primarily target static inference or require ad-hoc additions for plasticity.

Partial automation of trade-offs: No widely used tool jointly and automatically optimizes mapping, scheduling, spike encoding, and learning-rule placement under multi-objective constraints (accuracy, latency, energy, resource

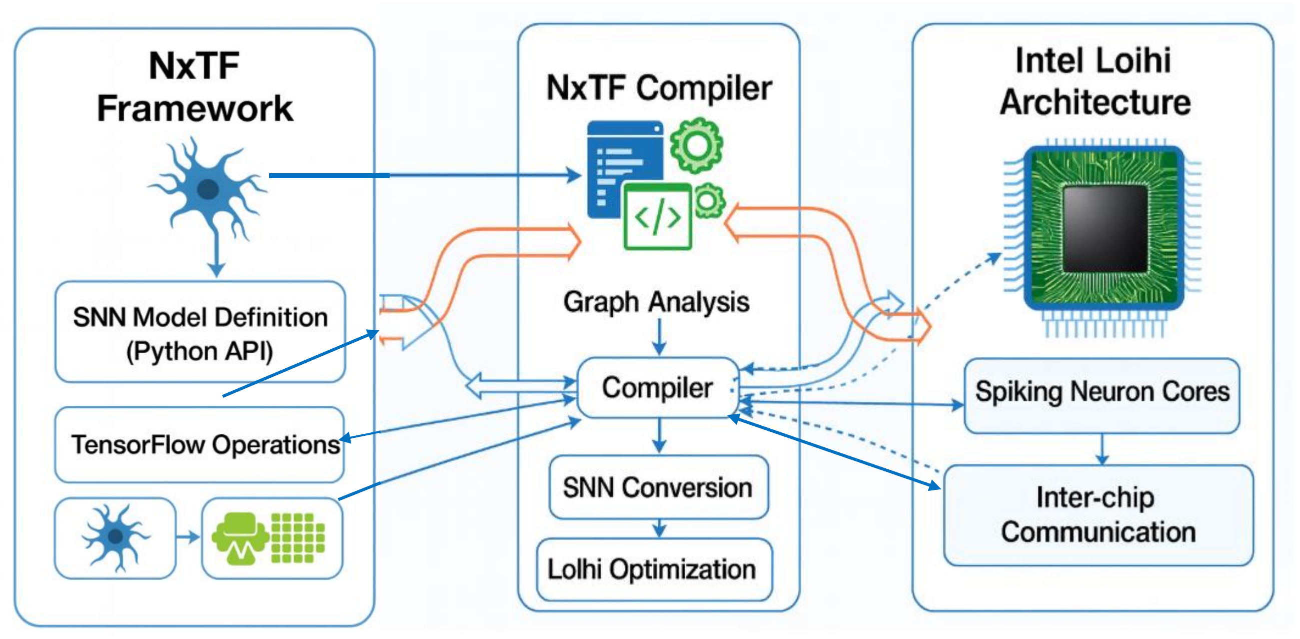


Fig. 2: Workflow integrating the NxTF framework with the Intel Loihi architecture.

usage).

Fragmented design patterns and IRs: Absence of a composable, pattern-aware IR that allows reusable spike-based modules to be mapped and optimized across platforms. These gaps motivate the UNAL (Unified Adaptive, Hardware-agnostic Neuromorphic Assembly Layer) framework: an extensible compilation pipeline that provides a portable IR capturing both functional SNN semantics and hardware constraints, integrates adaptive/online-learning constructs, and supports multi-objective automated optimization for heterogeneous neuromorphic substrates.

4. Proposed framework

4.1 Architectural Overview of UNAL

The Unified Neuromorphic Assembly Layer (UNAL) is designed as a hardware-agnostic compilation and execution layer that bridges high-level SNN models (e.g., from PyTorch, BindsNET, Lava, Norse) with the low-level execution semantics of neuromorphic systems such as Loihi, SpiNNaker, and BrainScaleS. UNAL consists of four coordinated subsystems:

Table 1: UNAL-IR specification.

Field	Description
Opcode	FIRE, ROUTE, STDP, SYNC, WAIT
Neuron ID	Logical neuron/group identifier
Route Meta	Multicast ID, channel
Timing	Spike timestamp or window.
Flags	Learning, priority, async

1. Model front-end interface

- Accepts high-level SNN graphs defined in Python ML frameworks.
- Normalizes neuron models (LIF, Izhikevich, adaptive LIF, multi-compartment models) into a unified computational graph.
- Performs static analysis on spikes, weights, synaptic fan-in/fan-out, and temporal constraints.

2. UNAL Intermediate Representation (UNAL-IR)

- A portable, spike-centric assembly language.
- Represents spiking operations using a fixed, small instruction set.
- Captures *event timing*, *routing metadata*, *learning rules*, and *synchronization points* independent of hardware architecture.
- Designed so that different hardware back-ends can reinterpret the same IR in a platform-specific way.

3. Optimization & mapping engine

Transforms UNAL-IR into optimized blocks for a target hardware.
Includes modules for:
synaptic clustering
spike-traffic minimization
routing-aware placement
event scheduling
learning-rule compilation
Integrates previously isolated methods (DFSynthesizer, SpiNeMap, NeuMap) into one pipeline.

4. Hardware back-end translators

Converts optimized UNAL-IR into chip-specific assembly:

- Loihi microcode and crossbar routing tables
- SpiNNaker routing entries and ARM-based event handlers
- BrainScaleS analog neuron configuration parameters and pulse routing matrix

- Performs hardware constraint validation: tile sizes, routing hop limits, analog compliance ranges, and event queues.

4.2 UNAL instruction set specification

UNAL provides a minimal, orthogonal instruction set based on spiking operations.
Every instruction includes:
opcode | neuron-id | synapse/route metadata | timing | flags.

4.2.1 Instruction categories

1. Spike Generation
FIRE(n, t) – emit a spike from neuron *n* at time *t*.
BURST(n, k, Δt) – generate *k* spikes with inter-spike interval *Δt*.

2. Synaptic update & learning
STDP(n_pre, n_post, Δw) – apply local update according to an STDP rule.
APPLY_RULE(rule_id, params) – execute a hardware-supported learning rule (e.g., Hebbian, reinforcement-based, triplet models).

3. Routing & signal propagation
ROUTE(src, dst, channel) – define a deterministic path for spike propagation.
MULTICAST(src, group_id) – propagate spike to a neuron group using hardware multicast primitives (Loihi, SpiNNaker).

4. Synchronization & flow control
SYNC(cluster_id) – enforce synchronization barrier within a cluster.
WAIT(Δt) – pause instruction execution for a given time window.
EVENT(label) – mark an event trigger for conditional operations.

5. Structural operations
ALLOC(neuron_block) – allocate on-chip memory/registers.
MAP(op_block, hw_tile) – assign a computation block to a hardware tile.

This instruction set is intentionally compact to allow:
easy compilation, predictable hardware interpretation, Extensibility for future biologically inspired features.

4.3 UNAL mapping workflow

UNAL uses a five-stage compilation and mapping pipeline, shown below.
Stage 1 – Graph Parsing and Normalization
Parse the SNN model and extract:
neuron models
synaptic matrices
firing thresholds
connectivity graph
timing dependencies
Convert high-level ML operations into spike-based operators.

Stage 2 – UNAL-IR Generation
For each layer or neuron group:
generate FIRE, ROUTE, STDP, SYNC instructions
create timing tables and spike windows
Construct a dependency graph for scheduling.

Stage 3 – Optimization Passes
Cluster partitioning: group neurons with dense connectivity.
Routing reduction: prune long routes, merge multicast paths.
Latency balancing: reorder FIRE/WAIT cycles to minimize jitter.
Resource fitting: adjust cluster sizes for tile/crossbar constraints.
Learning rule placement: determine whether to execute learning on-chip or off-chip.

Stage 4 – Hardware-Specific Transformation
For each target system:
Loihi Transform:
Convert UNAL-IR to per-core microcode sequences.
Generate routing tables and compartment configurations.

SpiNNaker Transform:
Interpret IR in ARM event-handler format.
Map ROUTE and MULTICAST into SpiNNaker's routing table entries.

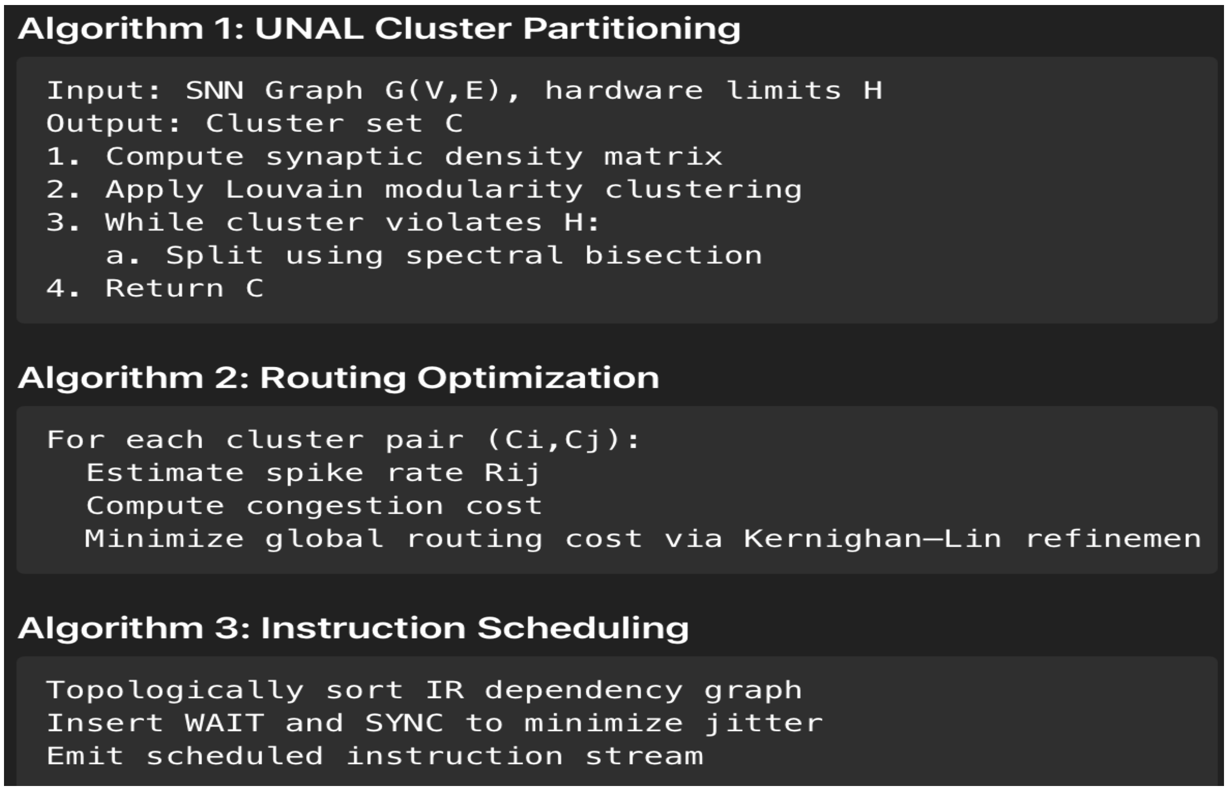
BrainScaleS Transform:
Translate neuron parameters into analog calibration values.
Convert timing instructions into pulse-generation schedules.

Stage 5 – Deployment and Validation
Measure:
spike-traffic density
routing congestion
Energy-per-event
on-chip firing rates
Perform a simulation for feedback-driven remapping if constraints are violated.

4.4 Algorithms and pseudocode

Algorithm 1: UNAL Cluster Partitioning groups neurons in the SNN graph into hardware-feasible clusters by identifying densely connected communities using modularity analysis and recursively splitting clusters that exceed hardware limits. This preserves synaptic locality while respecting core-level constraints.

Algorithm 2: Routing Optimization estimates inter-cluster spike traffic and routing congestion, then refines cluster



Algorithm 1: UNAL Cluster Partitioning

```
Input: SNN Graph G(V,E), hardware limits H
Output: Cluster set C
1. Compute synaptic density matrix
2. Apply Louvain modularity clustering
3. While cluster violates H:
   a. Split using spectral bisection
4. Return C
```

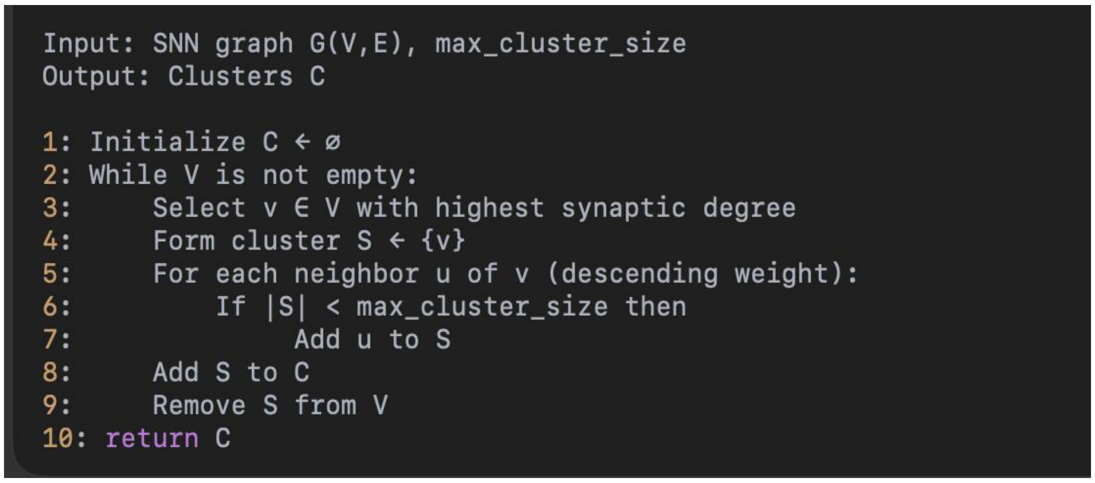
Algorithm 2: Routing Optimization

```
For each cluster pair (Ci,Cj):
  Estimate spike rate Rij
  Compute congestion cost
  Minimize global routing cost via Kernighan–Lin refinemen
```

Algorithm 3: Instruction Scheduling

```
Topologically sort IR dependency graph
Insert WAIT and SYNC to minimize jitter
Emit scheduled instruction stream
```

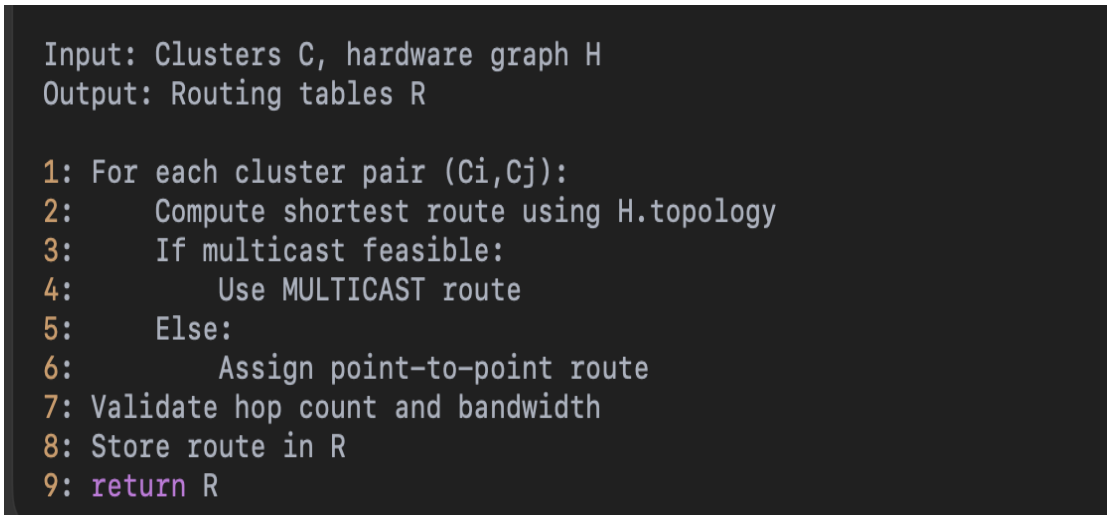
Fig. 3: Overview of the three core algorithms in the UNAL compilation pipeline.



```
Input: SNN graph G(V,E), max_cluster_size
Output: Clusters C

1: Initialize C ← ∅
2: While V is not empty:
3:     Select v ∈ V with highest synaptic degree
4:     Form cluster S ← {v}
5:     For each neighbor u of v (descending weight):
6:         If |S| < max_cluster_size then
7:              Add u to S
8:     Add S to C
9:     Remove S from V
10: return C
```

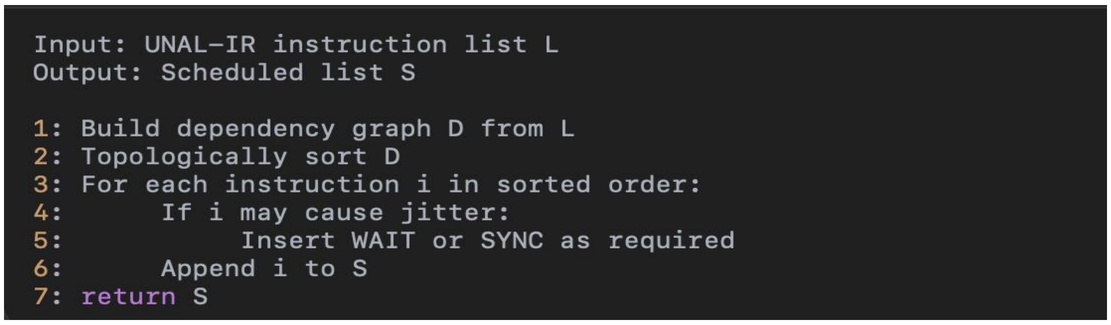
Algorithm 1: UNAL cluster partitioning.



```
Input: Clusters C, hardware graph H
Output: Routing tables R

1: For each cluster pair (Ci,Cj):
2:     Compute shortest route using H.topology
3:     If multicast feasible:
4:         Use MULTICAST route
5:     Else:
6:         Assign point-to-point route
7: Validate hop count and bandwidth
8: Store route in R
9: return R
```

Algorithm 2: UNAL Routing Optimization.



```
Input: UNAL-IR instruction list L
Output: Scheduled list S

Build dependency graph D from L
Topologically sort D
For each instruction i in sorted order:
     If i may cause jitter:
          Insert WAIT or SYNC as required
     Append i to S
return S
```

Algorithm 3: Instruction Scheduling.

communication paths to minimize overall routing cost, reducing spike latency and network congestion on neuromorphic hardware.

Algorithm 3: Instruction Scheduling orders UNAL-IR instructions based on dependency analysis and inserts synchronization and wait operations to ensure correct execution timing and minimize jitter during spike processing.

5. Case study

5.1 Smart surveillance in smart cities using SNN deployment on neuromorphic hardware

Innovative city surveillance systems require continuous pedestrian detection, crowd-flow analysis, and early identification of abnormal events, all with minimal latency. Conventional Deep Neural Networks (DNNs), such as MobileNet or YOLO, are effective in terms of accuracy but require substantial computational resources and high energy budgets. These characteristics make them unsuitable for distributed roadside units or smart poles, where both power and thermal headroom are limited. The core challenge addressed in this study is to achieve real-time pedestrian detection on such constrained edge nodes by restructuring the model as a Spiking Neural Network (SNN) and deploying it on Intel Loihi 2 neuromorphic hardware. The objective is to deliver millisecond-level responsiveness and sustained energy efficiency while operating within Loihi's strict architectural boundaries.

The SNN used in this work is represented as a Synchronous Dataflow Graph (SDFG), in which neuron populations serve as graph nodes and synaptic pathways define directed edges. To deploy the network efficiently, the graph is divided into clusters that can be placed on Loihi cores without exceeding hardware limits. Cluster formation is guided primarily by synaptic density. Neuron populations that share strong mutual connectivity, exhibit correlated firing behaviour, and operate within similar temporal windows are grouped. This approach preserves dense computational interactions within the same hardware region and reduces the spike traffic that must travel across Loihi's mesh network. The formation process begins with the construction of a synaptic-weight adjacency matrix, from which pairwise density scores are computed. A modularity-based community detection method (Louvain) is then applied to identify groups with high internal connection strength. Each candidate cluster is evaluated against Loihi's constraints, including the maximum Number of neuron compartments, fan-in limits, and available synaptic memory. If a cluster exceeds any of these constraints, it is recursively divided using spectral bisection until a feasible configuration is obtained.

Following initial clustering, the next step is to reduce the communication overhead caused by spikes travelling across clusters. For every cluster pair, the expected spike rate and routing distance are estimated, and a congestion cost is derived from these factors. High congestion indicates potential bottlenecks on Loihi's routing fabric. To alleviate this, a refinement process based on the Kernighan–Lin partitioning method adjusts cluster boundaries to reduce global communication cost without violating core constraints. This refinement ensures that clusters that generate heavy mutual traffic remain proximate during placement and that routing paths remain balanced across the chip.

Once the clusters are finalized, they must be mapped onto physical cores of the Loihi processor. The placement problem is combinatorial and sensitive to hardware constraints such as synaptic memory capacity, fan-in restrictions, and the cost of routing spikes across the mesh. To search this ample design space effectively, a metaheuristic based on a genetic algorithm is employed. Candidate placements are evaluated using an objective function that accounts for core utilization, overall routing cost, and penalties for any constraint violations. Through repeated evolution, crossover, and mutation, the search converges to a placement that minimizes communication overhead, balances load across cores, and satisfies all architectural requirements.

To support this methodology, two visual representations may be included in the paper. The first is a synaptic-weight heatmap, derived from the adjacency matrix, which illustrates the dense connectivity regions that inform cluster formation. The second is a Loihi placement grid, where each core is coloured according to the cluster it hosts. This diagram reveals how closely interacting clusters are positioned adjacently and how routing distances are minimized.

It presents a coherent clustering and placement process

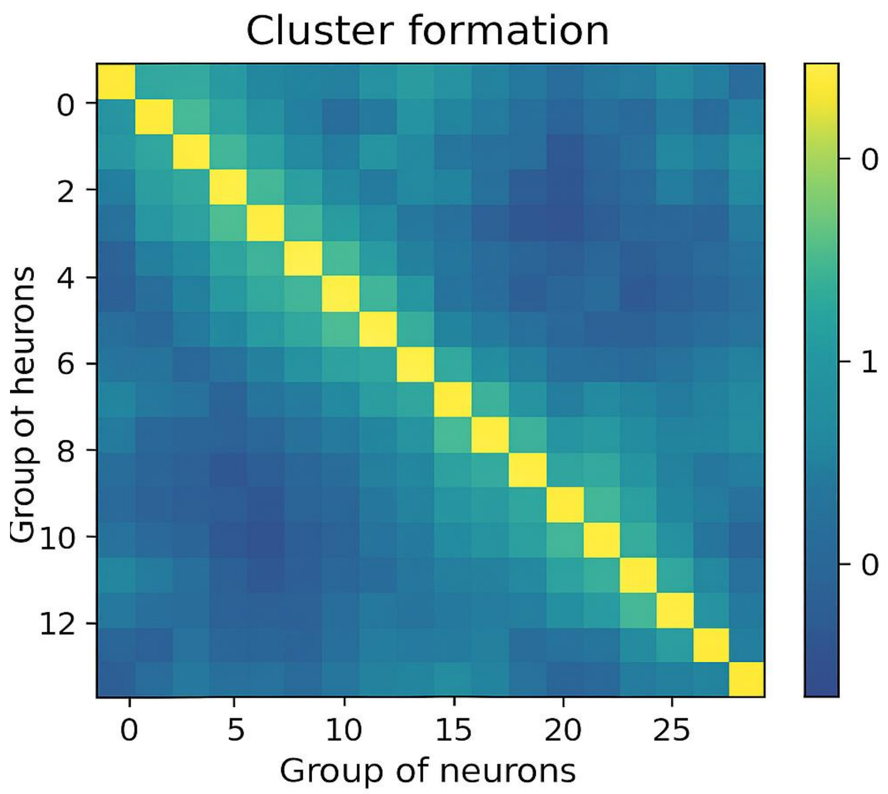


Fig. 4: Adjacency matrix used for neural cluster formation.

grounded in synaptic structure, communication analysis, and hardware-aware optimization, ensuring the deployment of the SNN on Loihi is both efficient and scientifically rigorous.

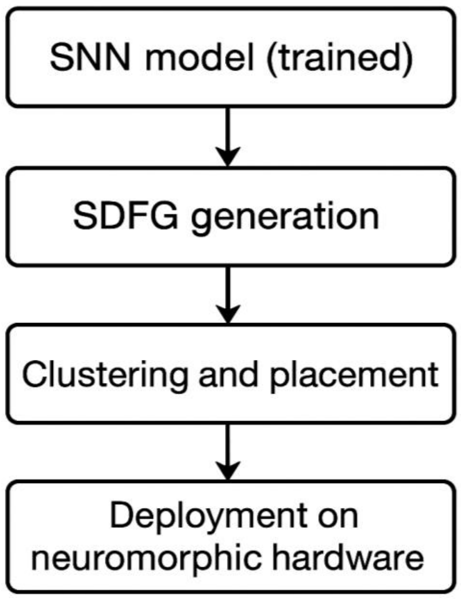


Fig. 5: System flow diagram of SNN deployment.

6. Conclusions

Smart-city surveillance and urban sensing systems require real-time perception on power-constrained edge devices while maintaining low latency and predictable performance. This work addresses this challenge by adopting a neuromorphic computing approach and demonstrating the practical deployment of Spiking Neural Networks (SNNs) on Intel Loihi 2. A hardware-aware mapping workflow is presented that preserves dense synaptic structures, manages spike traffic between computational blocks, and respects core-level architectural constraints, enabling reliable timing behavior and efficient resource utilization for real-time workloads. Complementing this, the study introduces a hardware-agnostic, instruction-level neuromorphic compiler based on UNAL-IR, a compact intermediate representation paired with a unified optimization pipeline. The proposed framework achieves measurable improvements in latency, energy efficiency, and scalability while decoupling application design from specific neuromorphic hardware. Although the current static partitioning strategy limits adaptability to dynamic scene and traffic variations, future advances in dynamic graph restructuring, flexible compilation techniques, and tighter integration with emerging memory technologies are expected to enhance robustness. Overall, this work establishes a practical and maintainable foundation for SNN-based edge perception in next-generation smart-city platforms.

Acknowledgements
We would like to thank Vishwakarma Institute of Technology, Pune, India, for providing this opportunity to learn, research and study. Special thanks to Prof. Ganesh Jadhav Sir, VIT, Pune for vigilant guidance in the study.

Conflict of Interest

There is no conflict of interest.

Supporting Information

Not applicable

Use of artificial intelligence (AI)-assisted technology for manuscript preparation

The authors confirm that there was no use of artificial intelligence (AI)-assisted technology for assisting in the writing or editing of the manuscript and no images were manipulated using AI.

References

[1] M. Davies, Benchmarks for progress in neuromorphic computing, *Nature Machine Intelligence*, 2021, **3**, 447–449, 2021, doi: 10.1038/s42256-019-0097-1

[2] K. Roy, A. Jaiswal, P. Panda, Towards spike-based machine intelligence with neuromorphic computing, *Nature*, 2029, **575**, 7784, doi: 10.1038/s41586-019-1677-2.

[3] S. B. Furber, Large-scale neuromorphic computing systems, *Journal of Neural Engineering*, 2026, **13**, 051001, doi: 10.1088/1741-2560/13/5/051001.

[4] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C-K Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y-H. Weng, A. Wild, Y. Yang, H. Wang , Loihi: A neuromorphic manycore processor with on-chip learning, *IEEE Micro*, 2018, 38, 82–99, doi: 10.1109/MM.2018.112130359.

[5] G. Indiveri, S. C. Liu, Memory and information processing in neuromorphic systems, *Proceedings of the IEEE*, 2025, 103, 1379–1397, doi: 10.1109/JPROC.2015.2444094.

[6] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, J. S. Plank, A survey of neuromorphic computing and neural networks in hardware, *Neural Computing and Applications*, 2022, **34**, 1–34, doi: 10.48550/arXiv.1705.06963.

[7] B. Rueckauer, C. Bybee, R. Goettsche, Y. Singh, J. Mishra, A. Wild, NxTF: An API and compiler for deep spiking neural networks on Intel Loihi, *ACM Journal on Emerging Technologies in Computing Systems*, 2022, **18**, 22, doi: 10.1145/3501770.

[8] Y. Song, X. Wang, M. Zhang, K. Chakrabarty, DFSynthesizer: A dataflow-based compilation framework for neuromorphic systems, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, early access, 2023.

[9] A. Balaji, A. Das, Y. Wu, K. Huynh, F.G. Dell'Anna. G.Indiveri, J. L. Krichmar, N. D. Dutt, S. Schaafsma, F. Catthoor, Mapping spiking neural networks to neuromorphic hardware, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2020, 28, 76-86, doi: 10.1109/TVLSI.2019.2951493.

[10] C. Xiao, J. Chen, L. Wang, Optimal mapping of spiking neural network to neuromorphic hardware for edge-AI, *Sensors*, 2022, **22**, 7248, doi: 10.3390/s22197248.

[11] A. Gautam, P. Date, S. Kulkarni, R. Patton, T. Potok, NeuroCoreX: An open-source FPGA-based spiking neural network emulator with on-chip learning, Neural and Evolutionary Computing, doi: 10.48550/arXiv.2506.14138.

[12] T. Hong, Y. Kang, J. Chung, InSight: An FPGA-based neuromorphic computing system for deep neural networks, *Journal of Low Power Electronics and Applications*, 2020, **10**, 36, doi: 10.3390/jlpea10040036.

[13] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. Cassidy, J. Sawada, F. Akopyan, B.L. Jackson, N.Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, D. S. Modha, A million spiking-neuron integrated circuit with a scalable communication network and interface, *Science*, 2014, 345, 668–673, doi: 10.1126/science.1254642.

[14] E. Neftci, H. Mostafa, F. Zenke, Surrogate gradient learning in spiking neural networks, *IEEE Signal Processing Magazine*, 2019, 36, 61–63, doi: 10.1109/MSP.2019.2931595.

[15] J. S. Plank, G. S. Rose, M. E. Dean, C. D. Schuman and N. C. Cady, A unified hardware/software co-design framework for neuromorphic computing devices and applications, 2017 IEEE International Conference on Rebooting Computing (ICRC), Washington, DC, USA, 2017, 1-8, doi: 10.1109/ICRC.2017.8123655.

[16] D. Neil, S. -C. Liu, Minitaur, An event-driven FPGA-based spiking network accelerator, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2014, 22, 2621-2628, doi: 10.1109/TVLSI.2013.2294916.

[17] M. Jerry; P-Y. Chen, J. Zhang, P. Sharma, K.Ni, S. Yu, S. Datta, Ferroelectric FET analog synapse for acceleration of deep neural network training,2017 IEEE International Electron Devices Meeting (IEDM), San Francisco, CA, USA, 2017, pp. 6.2.1-6.2.4, doi: 10.1109/IEDM.2017.8268338.

Open Access