aigverse: A Unified Infrastructure for Machine Learning-Driven Logic Synthesis

Marcel Walter
Chair for Design Automation
Technical University of Munich
MQSC
Munich, Germany
marcel.walter@tum.de

Isabella Venancia Gardner
Informatics Institute
University of Amsterdam
Amsterdam, Netherlands
bella.gardner@student.uva.nl

Robert Wille
Chair for Design Automation
Technical University of Munich
MQSC
Munich, Germany
robert.wille@tum.de

https://github.com/marcelwa/aigverse

Abstract

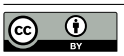
The rapid advancement of *Machine Learning* (ML) has demonstrated immense potential for logic synthesis, design-space exploration, and circuit optimization. In practice, however, researchers who want to combine these areas repeatedly face the same infrastructure problem: the surrounding ML and data-science ecosystem is centered on Python, whereas the core data structures and algorithms of logic synthesis are the product of decades of highly optimized C/C++ engineering. Connecting these environments often requires project-specific circuit conversions, tool orchestration, and data-preparation code. To address this recurring gap, this paper introduces *aigverse*, a unified open-source infrastructure project that brings mature logic synthesis capabilities into Python-first workflows without reimplementing them there. To this end, *aigverse* offers direct bindings to high-performance C/C++ synthesis libraries through an idiomatic Python interface. It provides reusable support for circuit construction and manipulation, benchmark access and dataset generation, optimization and equivalence-checking flows, and export into graph and tensor representations for downstream data science and ML pipelines. In a case study comparing a pure-Python tensorization pipeline from the literature against *aigverse*, the latter achieves up to 17.2× faster end-to-end tensorization while reducing isolated framework hand-off overhead by three to four orders of magnitude depending on circuit size. In this way, *aigverse* provides logic synthesis with the kind of reusable, domain-aware software bridge that has already helped ML become effective in scientific fields such as medicine and chemistry without requiring those researchers to become application-domain experts.

CCS Concepts

• **Hardware** → **Combinational synthesis**; • **Software and its engineering** → **Software libraries and repositories**; • **Computing methodologies** → *Neural networks*.

Keywords

Logic Synthesis, Machine Learning, Open Source, Electronic Design Automation, Graph Neural Networks, And-Inverter Graphs



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASPDAC '27, Tokyo, Japan
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2923-2/2027/01
https://doi.org/10.1145/3840404.3848264

ACM Reference Format:
Marcel Walter, Isabella Venancia Gardner, and Robert Wille. 2027. aigverse: A Unified Infrastructure for Machine Learning-Driven Logic Synthesis. In *32nd Asia and South Pacific Design Automation Conference (ASPDAC '27), January 25–28, 2027, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3840404.3848264

1 Introduction

As the scale and complexity of digital designs continue to accelerate, logic synthesis and optimization remain the foundational steps for translating high-level Boolean specifications into efficient, manufacturable hardware [6, 12, 30]. This pressure is compounded by a widening gap between hardware and compute scaling: transistor-level scaling has slowed markedly [20], even as machine learning compute demand has grown far faster, roughly doubling every six months since 2010 versus Moore's Law's historical multi-year cadence [35]. Mature logic synthesis tools and libraries rely on C/C++ implementations refined over decades for execution speed and scalability [5, 38, 44]. Python interfaces such as *Pyosys* [41] already make some synthesis functionality accessible from high-level development environments. The remaining challenge is to compose circuit objects, synthesis operations, and ML-facing data representations into reusable workflows, rather than implementing the connections anew for each application.

At the same time, logic synthesis workflows are increasingly studied, evaluated, and deployed in Python-centric contexts. This is particularly true in ML and data-science settings, where data preparation, model development, experimentation, and downstream analysis are commonly carried out in Python. In those contexts, researchers need reliable access to mature logic synthesis functionality accessible from within the Python ecosystem.

The intersection of these two domains is no longer theoretical. Recent work has repeatedly demonstrated the potential of learning-based approaches for logic synthesis, design-space exploration, and circuit optimization [28, 40, 43, 48]. Yet the practical software bridge between them has remained weak. Even with existing Python interfaces, individual applications may still need custom code for circuit manipulation, external-tool orchestration, dataset preparation, and conversion into graph or tensor formats. Reimplementing these connections duplicates engineering effort and can introduce avoidable process, serialization, and Python-object construction costs.

To this end, this paper introduces the *aigverse* infrastructure project, which offers native C/C++ synthesis algorithms and data

structures within a common Python workflow. Direct bindings provide in-memory network manipulation, optimization, and verification, while an optional *ABC* [5] bridge handles external-tool execution and circuit interchange for the caller. Benchmark loading, circuit generation, and graph- and tensor-based export compose with both paths. The resulting representations integrate with *NumPy* [19], *PyTorch* [33], *TensorFlow* [1], *NetworkX* [18], and *JAX* [4], without making these downstream frameworks mandatory dependencies of the synthesis core. On identical corpora, the native tensorization path yields up to 17.2× end-to-end speedups over a pure-Python baseline while cutting isolated tensor hand-off costs by three to four orders of magnitude as circuits scale.

We do not present *aigverse* as a complete answer to every EDA–ML integration problem; rather, we position it as a practical infrastructure contribution that closes a longstanding gap between high-performance synthesis tooling and Python-first workflows. Its contribution is the combination of AIG-oriented circuit construction and manipulation, native synthesis operations, reusable ABC orchestration, benchmark access, and direct graph and tensor interoperability within a single Python library.[1]

Specifically, this paper makes the following contributions.

(1) the introduction of *aigverse* as a unified infrastructure project that bridges high-performance logic synthesis and Python-first workflows;
(2) an architectural overview of the *aigverse* framework, demonstrating its dual utility as both a lean scripting tool and an ML-facing integration layer;
(3) usage examples that compose native algorithms, ABC recipes, benchmark access, and modern graph and tensor tooling; and
(4) a *Circuit Transformer* [28] case study that combines a qualitative infrastructure comparison with quantitative tensor-export measurements, showing how reusable logic-synthesis infrastructure reduces both engineering overhead and runtime cost at the ML boundary.

2 Preliminaries & Related Work

We first introduce AIGs and their synthesis operations in Section 2.1, review the open-source EDA ecosystem in Section 2.2, and summarize the ML workflows motivating this infrastructure in Section 2.3.

2.1 And-Inverter Graphs (AIGs)

In modern EDA, efficiently representing and manipulating vast Boolean functions is a foundational challenge, as traditional representations such as truth tables, *Sum-of-Products*, and *Binary Decision Diagrams* [8] become intractable at scale due to their exponential memory footprints. Consequently, *And-Inverter Graphs* (AIGs) have emerged as a scalable, compact, and highly dominant data structure in contemporary logic synthesis workflows [23, 24, 31, 32] and serve as the primary underlying data structure within *aigverse*. Formally, an AIG is a homogeneous *Directed Acyclic Graph* (DAG) whose node set is strictly limited to primary inputs and two-input logical AND nodes, with *inversion tags* on edges denoting logical negations without requiring distinct inverter nodes. Primary outputs are pointers to specific nodes, optionally carrying an inversion tag. For example, the 2-bit ripple-carry adder in Figure 1 computes the arithmetic addition $[c, s_1, s_0] = [a_1, a_0] + [b_1, b_0]$ using only 10 AND nodes.

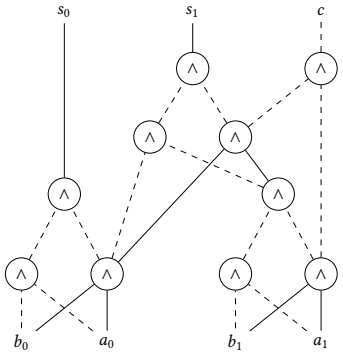


Figure 1: AIG of a 2-bit ripple-carry adder computing arithmetic addition: $[c, s_1, s_0] = [a_1, a_0] + [b_1, b_0]$.

Because AIGs are non-canonical, practical synthesis flows rely on a broad family of optimization and verification procedures to improve area, delay, and related cost metrics while preserving Boolean functionality, including *rewriting* [3, 32], *refactoring* [30, 32], *resubstitution* [12, 26, 30], and *equivalence checking* [23, 24, 31]. While effective, these procedures require rapid graph traversals, bitwise manipulation of truth tables, and dynamic memory allocation—a computational profile that has made heavily optimized low-level implementations, e. g. in C/C++, the practical foundation of modern logic synthesis.

2.2 The Open-Source EDA Ecosystem

Chief among the open-source logic synthesis tools is ABC [5], the academic and industrial gold standard for (sequential) synthesis, optimization, and verification for nearly two decades, complemented by *Yosys* [44], an open-source RTL synthesis framework, and, more recently, *mockturtle* [38], a C++17 header-only logic network library offering a generic, modular architecture for implementing synthesis data structures and algorithms. These tools share a relentless focus on execution speed and memory efficiency, leveraging native memory management, pointer-based traversals, and cache-friendly programming to handle graphs with millions of nodes.

Python-facing synthesis infrastructure already supports some of these workflows. *Pyosys* [41] exposes Yosys's C++ API to Python, including access to the design database and its modules, cells, and wires, and supports interactive inspection, modification, and development of custom passes. Other libraries serve complementary purposes: *py-aiger* [42] supports Python-side AIG construction and simulation, while *CircuitGraph* [39] represents circuits as *NetworkX* graphs for analysis and transformation.

This paper introduces *aigverse* as a complementary infrastructure effort focused on making high-performance logic synthesis a reusable part of the Python ecosystem, with native access to circuit

[1]The *aigverse* source code is publicly available at `https://github.com/marcelwa/aigverse`, with documentation at `https://aigverse.readthedocs.io`. Precompiled binaries for all major operating systems are distributed via PyPI and can be quickly installed using `uv pip install aigverse` (or standard `pip`).

objects and algorithms and direct integration with data-science and ML workflows.

2.3 Machine Learning on AIGs

Graph Neural Networks (GNNs) are a class of neural networks tailored to operate directly on graph-structured data via *message passing* [7, 16, 34], iteratively aggregating and updating node representations from their local neighborhood. An AIG maps cleanly onto this input format: node features can encode node type, logic level, and fan-out, while edge features capture logical inversions—an alignment that directly motivates the graph and tensor export capabilities of *aigverse* described in Section 3.3. Building on this alignment, learning-based EDA workflows increasingly use such representations for predictive quality-of-results estimation [9, 14, 27, 36, 37, 46, 47], for reinforcement-learning-driven optimization that navigates the exponential space of synthesis recipes [21, 25, 29], and for generative models that synthesize AIGs directly from functional specifications [28, 45, 48].

Despite their distinct objectives, these workflows share a recurring infrastructure requirement: circuit construction, optimization, verification, and feature extraction must compose with a Python-side learning loop. Without reusable support, researchers repeatedly implement network representations, file interchange, and external-tool orchestration themselves. These tasks impose engineering costs, while repeated process launches and serialization can also introduce latency. *aigverse* addresses both concerns: native operations avoid external-process and file-transfer overhead, whereas its ABC bridge encapsulates that overhead when established external recipes are required. The common Python interface lets researchers choose the appropriate execution path without rebuilding the surrounding workflow.

3 The Proposed Infrastructure: *aigverse*

The main contribution of this work is *aigverse*, an infrastructure project that makes logic synthesis and optimization available as reusable components in modern Python workflows with a direct path into data-science and machine-learning tooling. We first outline the project's architectural design in Section 3.1, then illustrate Pythonic workflows in Section 3.2, discuss integration with graph and tensor ecosystems in Section 3.3, and finally describe the software-engineering choices that support collaborative extension in Section 3.4.

3.1 Overview and Architectural Philosophy

The *aigverse* project combines two complementary execution paths within one Python workflow: direct bindings to the EPFL logic synthesis libraries [38] and reusable orchestration of an externally installed *ABC* executable [5]. Both accept network objects and compose with generation, benchmark access, verification, and ML-facing export. The native path avoids process and file-transfer costs; the ABC path exposes established recipes without requiring each application to implement its own tool interface.

Figure 2 summarizes this architecture: the compiled synthesis core is exposed through bindings, while the Python layer connects it to optional ABC execution, benchmark access, and downstream ecosystem integrations.

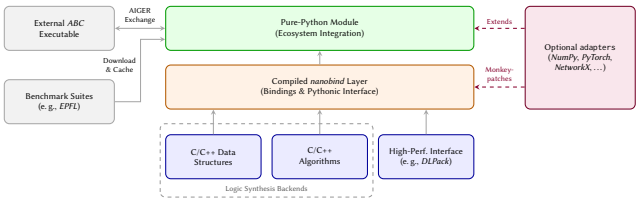


Figure 2: Layered architecture of *aigverse*. Core logic synthesis backends and performance-critical interfaces reside in compiled C/C++ code (bottom, blue). These are exposed to Python via *nanobind* bindings (center, yellow). A pure-Python module facilitates integration with the ecosystem (top, green). Heavier ML dependencies are loaded optionally via the `adapters` extra (right, red), which dynamically augments functionality across both the pure-Python and *nanobind* boundaries. The external ABC executable (left) connects to the Python module through the optional ABC bridge. Benchmark suites, such as EPFL (lower left), supply circuits through on-demand loading and local caching.

At the core, *mockturtle* provides network data structures and synthesis algorithms, *kitty* provides packed truth tables and symbolic simulation, and *lorina* provides circuit-format parsers. Five compiled *nanobind* [22] modules expose networks, algorithms, I/O, generators, and utilities as Python objects and functions. Graph-tensor export is implemented alongside the logic network bindings, so downstream frameworks can consume contiguous CPU arrays through *DLPack* without reconstructing graph features in Python.

The pure-Python layer provides benchmark loading, ABC orchestration, and ecosystem integration. Importing `aigverse.adapters` adds `Aig.to_networkx()` through runtime monkey-patching; its optional installation extra supplies *NumPy* and *NetworkX*. No heavy consumer dependencies such as *PyTorch* are installed automatically, keeping the library lean. ABC is also optional and is not bundled, allowing users to provide their custom binary: the bridge locates an installed executable, transfers the input as binary AIGER, executes the requested commands in an isolated working directory, and reads the result into a network object. This path retains subprocess and file-transfer costs while making their implementation reusable.

3.2 Pythonic Workflows

Networks and algorithms are ordinary Python symbols that compose in scripts, notebooks, and analysis pipelines. Users can construct circuits directly, apply native optimization passes, or execute ABC recipes on the same network objects. The following example loads a standard benchmark, applies native cut rewriting, composes a parameterized ABC schedule, and verifies the result:

```python
from aigverse import abc
from aigverse.algorithms import aig_cut_rewriting, equivalence_checking
from aigverse.benchmarks import epfl
original = epfl("ctrl")
native = aig_cut_rewriting(original, cut_size=4, preserve_depth=True)
recipe = [
    abc.balance.cmd(duplicate_critical=True),
    abc.rewrite.cmd(zero_cost=True),
    abc.refactor.cmd(max_support=10),
]
optimized = abc.run_script(native, recipe, timeout=60)
assert equivalence_checking(original, optimized)
```

The benchmark loader fetches circuits from the EPFL combinational suite [2] on demand and caches each revision locally. Its default revision is pinned, so independent experiments load the same circuits.

The `aigverse.abc` namespace provides named recipes such as `resyn2` and `compress2rs`, individual transformations, and arbitrary scripts. Each command wrapper's `.cmd()` method builds a parameterized command without executing it, allowing an entire schedule to run in one ABC process. Canonical recipe expansions are supplied by *aigverse*, avoiding dependence on local `abc.rc` aliases by default. The `aigverse.abc.gia` namespace exposes ABC's separate GIA command store, and `run_many` executes one schedule over a corpus with bounded concurrency and results in input order. Timeouts and structured execution errors make failures visible to the experiment rather than requiring custom output handling. Structured ABC statistics and a separate equivalence-checking interface also support comparison and validation across execution paths.

This supports experiments such as the repository's recipe study, which composes transformations, sweeps their order over EPFL circuits, and compares the resulting gate counts and depths.

Truth tables for outputs and internal nodes are available through symbolic simulation and *kitty*'s packed representations. Register-bearing circuits are supported as well, with cycle-based simulation and ABC transfer through AIGER latches.

3.3 Integration with Machine Learning

The previously presented workflows are foundational, but the most distinctive aspect of *aigverse* is that it integrates natively with established data-science and ML libraries and turns the graph and tensor representations discussed in Section 2.3 into first-class outputs of the synthesis stack. The features discussed in this section are representative rather than exhaustive: they illustrate a graph-analytics path through *NetworkX* and a lower-overhead tensor path through *DLPack* [13] and *PyTorch Geometric* [15].

NetworkX is a natural first integration target because it offers a well-established graph abstraction for visualization, structural analysis, feature mining, and downstream hand-off into graph-learning workflows. In *aigverse*, this path is enabled through the optional `adapters` extra: as discussed above, importing `aigverse.adapters` augments the module with auxiliary functions such as `Aig.to_networkx()`.

For example, *NetworkX* can extract the fan-in cone of each primary output for localized structural analysis:

```python
import networkx as nx
import aigverse.adapters

graph = original_aig.to_networkx()
outputs = [n for n, attrs in graph.nodes(data=True)
           if attrs["type"][3]]
cones = {
    po: graph.subgraph(nx.ancestors(graph, po) | {po}).copy()
    for po in outputs
}
```

For high-throughput training pipelines, however, materializing an intermediate *NetworkX* object can still impose unnecessary Python-side overhead, especially for large designs or real-time settings such as inference or RL loops. For such cases, *aigverse* also supports direct tensor export via *DLPack*, a cross-framework tensor exchange protocol that packages tensor memory, shape, type, and device metadata into a zero-copy transferable capsule that can be consumed by another framework through a corresponding `from_dlpack()` entry point. In practice, this means that tensor-producing code in one environment can hand off graph data to consumers such as *NumPy*, *PyTorch*, *TensorFlow*, or *JAX* without data copy overhead. In the current interface, `Aig.to_graph_tensors()` returns the graph connectivity in COO form together with node-wise and edge-wise feature tensors whose encodings can be configured and, optionally, enriched with derived structural information such as level, fanout, or truth-table data.

```python
import torch
from torch_geometric.data import Data

from aigverse.networks import EdgeTensorEncoding, NodeTensorEncoding

tensors = aig.to_graph_tensors(
  node_encoding=NodeTensorEncoding.ONE_HOT,
  edge_encoding=EdgeTensorEncoding.SIGNED,
  levels=True,
  fanouts=True,
)

edge_index = torch.from_dlpack(tensors["edge_index"])
edge_attr = torch.from_dlpack(tensors["edge_attr"])
node_attr = torch.from_dlpack(tensors["node_attr"])

data = Data(x=node_attr, edge_index=edge_index, edge_attr=edge_attr)
```

These tensors map directly to the *PyTorch Geometric* data abstraction: `x` contains node features, `edge_index` stores COO connectivity, and `edge_attr` captures edge-level information such as inversion polarity. The *DLPack* hand-off avoids an additional reconstruction step at the framework boundary, which is particularly attractive for batched training pipelines and low-latency interaction loops; Section 4 isolates this effect quantitatively. More generally, instead of forcing synthesis results through external translation pipelines, *aigverse* tightly integrates synthesis, graph extraction, and data preparation within a single infrastructure stack.

3.4 Extensibility and Community Benefit

The feature discussion in the previous sections is illustrative rather than exhaustive, but it already points to the broader role of *aigverse*. Its contribution is to make logic synthesis and optimization first-class citizens in the Python ecosystem by exposing them through objects, functions, protocols, and interfaces that integrate naturally with modern scientific software.

This, in turn, makes *aigverse* valuable both as an immediately useful tool and as a collaborative substrate. Researchers primarily interested in synthesis can use it as a lightweight scripting and experimentation layer; researchers interested in data science and ML can leverage its ecosystem integration and build directly on its graph and tensor interfaces. That broader role depends not only on exposed functionality but also on the surrounding engineering choices that make the project reusable in practice: typed interfaces and generated stubs improve editor support, API discoverability, and type safety; cross-platform CI testing, pre-built wheels, and Stable-ABI distribution make the package easier to install and use across operating systems and Python versions without requiring users to build the compiled layers themselves; and contributor-facing documentation, developer guides, and automated quality checks make it easier to extend the library without fragmenting it. The compiled

extensions use *nanobind*'s split mode, targeting the CPython 3.10 Stable ABI (`abi3`) while resolving the interpreter-specific binding runtime through `nanobind-backend`. Free-threaded support targets Python 3.15+ through the separate `abi3t` Stable ABI, enabling parallel Python execution without the GIL.

The public source repository and documentation expose these interfaces and usage examples for reuse. These choices help make new backends, bindings, encodings, adapters, and higher-level workflows incrementally contributable, so that *aigverse* is best understood less as a finished endpoint and more as a practical foundation for a broader open ecosystem. The practical relevance of that claim is easiest to see by the example of *Circuit Transformer* [28], a recent ML implementation that had to rebuild much of this surrounding infrastructure, as discussed in the following section.

4 Case Study: Circuit Transformer

Circuit Transformer [28] constitutes a useful case study of *aigverse*'s potential because its main scientific contribution is a learning model, yet its public implementation still required substantial code overhead to support that model. This is not a criticism of that work; rather, it reflects a challenge in the software landscape that researchers face when using logic synthesis within Python-based ML workflows. We use this example in two complementary ways: first, as a qualitative infrastructure comparison in Section 4.1, and second, as a quantitative benchmark for the ML-facing tensorization path in Section 4.2.

4.1 Qualitative Infrastructure Contrast

In the reference implementation available on GitHub,[2] *Circuit Transformer*'s auxiliary `utils.py` module defines custom classes for AIGs, implements AIGER parsing and serialization, computes truth tables via Python bit arrays, and invokes a pre-compiled *ABC* binary through temporary files for structural hashing, optimization, and equivalence checking. With over 30 % of the entire project's code attributed to this utility overhead, a substantial amount of engineering effort is spent not on the transformer itself, but on reconstructing the surrounding logic synthesis support that the learning workflow depends on. Table 1 summarizes this contrast at the level of the concrete workflow components.

The core issue is the volume of boilerplate code required to connect these ecosystems. In *Circuit Transformer*, custom AIG data structures, truth-table utilities, file-format handling, and repeated tool orchestration must be implemented before the model can function. In *aigverse*, these tasks become reusable library operations.

The significance of this reduction extends far beyond eliminating a single system call to ABC or bypassing filesystem interactions. Rather, a broad slice of the auxiliary infrastructure required by a modern ML project for logic synthesis can now be treated as library functionality. This dismantles a major technical barrier that has historically isolated EDA from mainstream machine learning: ML experts do not have to additionally become logic synthesis experts to apply their domain knowledge. This part of the case study, therefore, does not suggest that *Circuit Transformer* should have been written differently with the tools available at the time. Its value is evidentiary: it shows that the infrastructure gap identified in this paper is real, current, and large enough to absorb a substantial fraction of the implementation effort around an otherwise separate model contribution.

[2] `https://github.com/snowkylin/circuit-transformer`

Table 1: *Circuit Transformer* components and corresponding *aigverse* support eliminating infrastructure burden.

***Circuit Transformer* component(s)**	**Infrastructure burden and corresponding *aigverse* support**
`Node`, `NodeWithInv`	**In-memory AIG representation.** *Circuit Transformer* introduces custom Python classes for AIG nodes and complemented edges, assembled into Python lists to represent AIGs. *aigverse* instead exposes a native `Aig` class backed by *mockturtle*, so circuit construction and manipulation are based on an existing high-performance C++ network data structure.
`structural_hashing()` with *ABC*	**Structural hashing.** *Circuit Transformer* delegates structural hashing to *ABC* and interfaces with it via temporary files that require disk I/O. In *aigverse*, structural hashing is already built into the underlying AIG representation, so structurally identical nodes are reused automatically during construction and manipulation.
`compute_input_tt()`, `compute_tt()`, `compute_tts()`	**Truth-table support.** *Circuit Transformer* implements truth-table simulation explicitly in Python using the `bitarray` data structure. *aigverse* instead provides a dedicated `TruthTable` class backed by *kitty*, implemented as C++ packed bit arrays; symbolic simulation via `simulate()`, `simulate_nodes()`, and related operations enables fast truth table extraction from AIGs.
`read_aiger()`, `write_aiger()`, `aigtoaig`	**AIGER interchange.** *Circuit Transformer* includes custom AIGER parsing and serialization code together with explicit format-conversion calls. *aigverse* exposes the same interchange path as library functionality through the *lorina* parsers and serializers and adds support for further established file formats, e. g., Verilog and PLA, on top of that.
`sequential_synthesis()`, `cec()`	**Optimization and equivalence checking.** *Circuit Transformer* orchestrates optimization and equivalence checking by writing temporary files, invoking *ABC*, and rereading the results. *aigverse* provides native in-memory *mockturtle* algorithms and an optional ABC bridge that handles recipe execution and AIGER exchange through reusable Python calls.
Additional glue code on top of the custom circuit representation	**ML-facing representations.** *Circuit Transformer* must add further Python code to prepare model-facing circuit data on top of its custom representation. *aigverse* provides graph and tensor export directly from the network object itself.

The following native example (reusing concepts from Section 3.3) illustrates the kind of support functionality that *Circuit Transformer* implements across its `utils.py` in 769 lines: it starts from an AIG, applies an optimization step, checks functional equivalence, extracts truth tables for internal nodes, and produces an ML-ready tensor representation, all without leaving Python, without exchanging

Table 2: Benchmark summary for three tensorization paths on a shared corpus of 100 AIGs per bucket. The last row lists the ratio of the *Circuit Transformer* (CT) [28] baseline over the *aigverse* + *DLPack* path.

Method	End-to-End [s]		Handoff [ms]	
	10 K	100 K	10 K	100 K
CT (pure Python) [28]	16.12	169.94	1578.12	16 468.95
aigverse + Python Tensors	8.15	82.34	1533.28	15 575.98
aigverse + *DLPack*	1.07	9.86	0.75	0.62
Speedup (over CT)	15.12×	17.23×	2.11×10^3	2.65×10^4

files with external tools, and with the performance of compiled C/C++ data structures engineered over decades.

```python
import torch
from torch_geometric.data import Data

from aigverse.algorithms import aig_cut_rewriting, equivalence_checking
from aigverse.generators import random_aig
from aigverse.networks import EdgeTensorEncoding, NodeTensorEncoding


aig = random_aig(num_pis=8, num_gates=100, seed=42)
optimized = aig_cut_rewriting(aig, cut_size=4)

assert equivalence_checking(aig, optimized)

tensors = optimized.to_graph_tensors(
  node_encoding=NodeTensorEncoding.ONE_HOT,
  edge_encoding=EdgeTensorEncoding.SIGNED,
  node_tts=True,
)
edge_index = torch.from_dlpack(tensors["edge_index"])
edge_attr = torch.from_dlpack(tensors["edge_attr"])
node_attr = torch.from_dlpack(tensors["node_attr"])

data = Data(x=node_attr, edge_index=edge_index, edge_attr=edge_attr)
```

4.2 Quantitative Tensorization Benchmark

To complement the qualitative comparison above, we benchmarked the ML-facing tensorization path on a shared corpus of AIGER files, generated once and reused by all methods. The benchmark covers two regimes: 10 K-node circuits (200 PIs and 9,800 AND nodes) and 100 K-node circuits (1,000 PIs and 99,000 AND nodes), with 100 circuits per bucket. We evaluate three distinct pipelines to progressively isolate the source of performance gains: (i) a *Circuit Transformer* pure-Python AIG loader together with Python tensor construction [28], (ii) *aigverse* as the backend logic network but with tensors constructed in Python, and (iii) *aigverse* fully utilizing compiled tensor export and *DLPack* zero-copy handoff. Each method is evaluated in isolated subprocesses, reporting the median of five timed runs after two warm-up iterations using a single Torch thread.

All experiments were conducted on a Manjaro Linux system equipped with an AMD Ryzen 7 PRO 6850U processor and 32 GB of DDR5 RAM. Node and edge features used one-hot encodings, with optional level and fanout features disabled.

Table 2 summarizes the results. At the handoff step—bridging the logic-synthesis data structures into the Python ML framework—the impact of native C++ tensors and *DLPack* is staggering. While the pure-Python baseline spends roughly 16.47 s in handoff alone for 100 K-node circuits, the natively compiled *aigverse* path requires just 0.62 ms, yielding an improvement of four orders of magnitude (2.65×10^4). Because the native path operates in roughly $O(1)$ relative to Python's object allocation overhead, the relative speedup scales progressively with circuit size. In isolation, the intermediate *aigverse* + Python Tensors track confirms that without *DLPack*, the framework boundary remains an indomitable bottleneck, regardless of the underlying logic network backend.

End-to-end, replacing the pure-Python baseline with *aigverse* employing Python tensors still roughly halves the runtime to 82.34 s (a 2.06× speedup) due to superior load and traversal efficiency. However, when native export and zero-copy handoff are fully utilized, the runtime collapses to 9.86 s, a 17.2× speedup over the *Circuit Transformer* baseline. Together with the qualitative comparison above, these measurements reinforce the same point: *aigverse* removes both the engineering burden of rebuilding logic-synthesis support in Python and, more specifically, turns the synthesis-to-ML handoff from a multi-second Python bottleneck into an almost negligible step.

5 Conclusion

Reusable infrastructure is increasingly important for connecting logic synthesis with Python-based scientific workflows. Researchers working across both domains too often rebuild access to circuit objects, synthesis operations, and graph and tensor representations for their data-science and ML pipelines. We introduced *aigverse* as a unified infrastructure project providing that bridge. Direct bindings to mature C/C++ synthesis libraries and reusable ABC orchestration make established synthesis functionality available within familiar Python environments. Its modular, lean-core architecture supports concrete workflows that researchers otherwise rebuild piecemeal, including circuit construction, optimization, dataset generation, and graph and tensor export for downstream analysis and learning.

A *Circuit Transformer* case study makes this concrete both qualitatively and quantitatively: substantial effort was spent by the original authors reimplementing AIG representations, truth-table utilities, and file-based tool orchestration around otherwise separate ML contributions, whereas *aigverse* facilitates the same workflows with a few library calls, yielding up to 17.2× faster tensorization and reducing handoff overhead by three to four orders of magnitude. We do not present *aigverse* as a final answer to every EDA–ML integration challenge. Its contribution is more pragmatic: reducing duplicated engineering effort, lowering the barrier to entry for Python-based research on logic circuits, and giving logic synthesis a reusable bridge library already common in computational biology and medical imaging [10, 11, 17]—so future projects can focus on the scientific question they set out to study.

Acknowledgments

Anthropic, OpenAI, and Google LLMs were used for assistance with wording and clarity, and to support coding and debugging tasks related to the proposed *aigverse* project. All such outputs were reviewed and verified by the authors, who assume full responsibility for the content of this paper.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al.

2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–283.

[2] L. Amarú, P.-E. Gaillardon, and G. De Micheli. 2015. The EPFL Combinational Benchmark Suite. In *International Workshop on Logic and Synthesis (IWLS)*.

[3] Per Bjesse and Arne Borälv. 2004. DAG-Aware Circuit Compression for Formal Verification. In *ICCAD*. 42–49.

[4] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. https://github.com/jax-ml/jax

[5] Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification*. Springer, 24–40.

[6] Robert K. Brayton. 1984. *Logic Minimization Algorithms for VLSI Synthesis*. Springer.

[7] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Velickovic. 2021. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges. *CoRR* abs/2104.13478 (2021). arXiv:2104.13478 https://arxiv.org/abs/2104.13478

[8] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (1986), 677–691. doi:10.1109/TC.1986.1676819

[9] Ye Cai, Rui Wang, Liwei Ni, Miao Liu, Xingyu Meng, Xiaoze Lin, Junfeng Liu, Biwei Xie, and Xingquan Li. 2026. AiLO: A Predictive Framework for Logic Optimization Using Multi-Scale Cross-Attention Transformer. *ACM Trans. Des. Autom. Electron. Syst.* 31, 4, Article 63 (March 2026), 28 pages. doi:10.1145/3757319

[10] M. Jorge Cardoso, Wenqi Li, Richard Brown, et al. 2022. MONAI: An Open-Source Framework for Deep Learning in Healthcare. *arXiv preprint arXiv:2211.02701* (2022).

[11] Peter J. A. Cock, Tiago Antao, Jeffrey T. Chang, et al. 2009. Biopython: Freely Available Python Tools for Computational Molecular Biology and Bioinformatics. *Bioinformatics* 25, 11 (2009), 1422–1423.

[12] Giovanni De Micheli. 1994. *Synthesis and Optimization of Digital Circuits* (1st ed.). McGraw-Hill Higher Education.

[13] DMLC. 2017. DLPack: Open In-Memory Tensor Structure. https://github.com/dmlc/dlpack.

[14] Faezeh Faez, Raika Karimi, Yingxue Zhang, Xing Li, Lei Chen, Mingxuan Yuan, and Mahdi Biparva. 2025. MTLSO: A Multi-Task Learning Approach for Logic Synthesis Optimization. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference, ASPDAC 2025, Tokyo, Japan, January 20-23, 2025*, Yuichi Nakamura and Yu Wang (Eds.). ACM, 72–78. doi:10.1145/3658617.3697721

[15] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[16] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, Vol. 70. 1263–1272.

[17] Richard J. Gowers, Max Linke, Jonathan Barnoud, et al. 2016. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. In *Proceedings of the 15th Python in Science Conference (SciPy)*. 98–105.

[18] Aric Hagberg, Pieter J. Swart, and Daniel A. Schult. 2008. Exploring Network Structure, Dynamics, and Function Using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy)*. 11–15.

[19] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362. doi:10.1038/s41586-020-2649-2

[20] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (2019), 48–60. doi:10.1145/3282307

[21] Abdelrahman Hosny, Soheil Hashemi, Mohamed Shalan, and Sherief Reda. 2020. DRiLLS: Deep Reinforcement Learning for Logic Synthesis. In *25th Asia and South Pacific Design Automation Conference, ASP-DAC 2020, Beijing, China, January 13-16, 2020*. IEEE, 581–586. doi:10.1109/ASP-DAC47756.2020.9045559

[22] Wenzel Jakob. 2022. nanobind: Tiny and Efficient C++/Python Bindings. https://github.com/wjakob/nanobind.

[23] A. Kuehlmann and F. Krohm. 1997. Equivalence Checking Using Cuts and Heaps. In *Design Automation Conference (DAC)*. 263–268.

[24] Andreas Kuehlmann, Vineet Paruthi, Florian Krohm, and Malay Ganai. 2002. Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 21, 12 (2002), 1377–1394. doi:10.1109/TCAD.2002.804386

[25] Walter Lau Neto, Yingjie Li, Pierre-Emmanuel Gaillardon, and Cunxi Yu. 2023. FlowTune: End-to-End Automatic Logic Optimization Exploration via Domain-Specific Multi-Armed Bandit. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2023).

[26] Siang-Yun Lee, Heinz Riener, Alan Mishchenko, Robert K Brayton, and Giovanni De Micheli. 2021. A Simulation-Guided Paradigm for Logic Synthesis and Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 8 (2021), 2573–2586.

[27] Min Li, Sadaf Khan, Zhengyuan Shi, Naixing Wang, Huang Yu, and Qiang Xu. 2022. DeepGate: learning neural representations of logic gates. In *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, Rob Oshana (Ed.). ACM, 667–672. doi:10.1145/3489517.3530497

[28] Xihan Li, Xing Li, Lei Chen, Xing Zhang, Mingxuan Yuan, and Jun Wang. 2025. Circuit Transformer: A Transformer That Preserves Logical Equivalence. (2025). arXiv:2403.13838 [cs.LG]

[29] Miao Liu, Liwei Ni, Junfeng Liu, Xingyu Meng, Rui Wang, Xiaoze Lin, Xinhua Lai, Xingquan Li, and Jungang Xu. 2026. A Survey of Machine Learning Approaches in Logic Synthesis. *ACM Trans. Design Autom. Electr. Syst.* 31, 2 (2026), 38:1–38:43. doi:10.1145/3785362

[30] Alan Mishchenko and Robert Brayton. 2006. Scalable Logic Synthesis using a Simple Circuit Structure. In *IWLS*, Vol. 6. 15–22.

[31] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. 2005. *FRAIGs: A Unifying Representation for Logic Synthesis and Verification*. Technical Report EECS-2005-72. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-72.html

[32] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. 2006. DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis. In *DAC*. 532–535.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8026–8037.

[34] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Trans. Neural Networks* 20, 1 (2009), 61–80. doi:10.1109/TNN.2008.2005605

[35] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. 2022. Compute Trends Across Three Eras of Machine Learning. In *2022 International Joint Conference on Neural Networks (IJCNN)*. arXiv:2202.05924 doi:10.1109/IJCNN55064.2022.9891914

[36] Zhengyuan Shi, Hongyang Pan, Sadaf Khan, Min Li, Yi Liu, Junhua Huang, Hui-Ling Zhen, Mingxuan Yuan, Zhufei Chu, and Qiang Xu. 2023. DeepGate2: Functionality-Aware Circuit Representation Learning. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–9. doi:10.1109/ICCAD57390.2023.10323798

[37] Zhengyuan Shi, Ziyang Zheng, Sadaf Khan, Jianyuan Zhong, Min Li, and Qiang Xu. 2024. DeepGate3: Towards Scalable Circuit Representation Learning. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. doi:10.1145/3676536.3676791

[38] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. De Micheli. 2019. The EPFL Logic Synthesis Libraries. arXiv:1805.05121v2

[39] Joseph Sweeney, Ruben Purdy, Ronald D. Blanton, and Lawrence Pileggi. 2020. CircuitGraph: A Python Package for Boolean Circuits. *Journal of Open Source Software* 5, 56 (2020), 2646. doi:10.21105/joss.02646

[40] Dimitrios Tsaras, Antoine Grosnit, Lei Chen, Zhiyao Xie, Haitham Bou-Ammar, and Mingxuan Yuan. 2024. ShortCircuit: AlphaZero-Driven Circuit Design. arXiv:2408.09858 [cs.LG]

[41] Benedikt Tutzer, Christian Krieg, Clifford Wolf, and Axel Jantsch. 2019. Python Wraps Yosys for Rapid Open-Source EDA Application Development. In *Workshop on Open-Source Design Automation (OSDA)*. Florence, Italy, 4 pages. https://osda.gitlab.io/19/3.3.pdf

[42] Marcell Vazquez-Chanlatte and Markus N. Rabe. 2018. py-aiger: A Python Library for Manipulating Sequential and Combinatorial Circuits Encoded Using And-Inverter Gates. https://github.com/mvcisback/py-aiger. doi:10.5281/zenodo.1405781

[43] Zhihai Wang, Jie Wang, Qingyue Yang, Yinqi Bai, Xing Li, Lei Chen, Jianye Hao, Mingxuan Yuan, Bin Li, Yongdong Zhang, et al. 2024. Towards Next-Generation Logic Synthesis: A Scalable Neural Circuit Generation Framework. *Advances in Neural Information Processing Systems* 37 (2024), 99202–99231.

[44] Clifford Wolf and Johann Glaser. 2013. Yosys – A Free Verilog Synthesis Suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*.

[45] Haoyuan Wu, Haisheng Zheng, Shoubo Hu, Zhuolun He, and Bei Yu. 2025. Architect of the Bits World: Masked Autoregressive Modeling for Circuit Generation Guided by Truth Table. *arXiv preprint arXiv:2502.12751* (2025).

[46] Haoyuan Wu, Haisheng Zheng, Yuan Pu, and Bei Yu. 2025. Circuit Representation Learning with Masked Gate Modeling and Verilog-AIG Alignment. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net. https://openreview.net/forum?id=US9k5TXVLZ

[47] Qiyun Zhao. 2025. FuncGNN: Learning Functional Semantics of Logic Circuits with Graph Neural Networks. *CoRR* abs/2506.06787 (2025). arXiv:2506.06787 doi:10.48550/ARXIV.2506.06787

[48] Xinyi Zhou, Xing Li, Yingzhao Lian, Yiwen Wang, Lei Chen, Mingxuan Yuan, Jianye Hao, Guangyong Chen, and Pheng Ann Heng. 2024. SeaDAG: Semi-autoregressive Diffusion for Conditional Directed Acyclic Graph Generation. arXiv:2410.16119 [cs.LG]