Journal of
Low Power Electronics and Applications

MDPI

Article

Versat-AI: An ONNX-to-SoC Compiler for Model-Agnostic CGRA Edge Inference

Rúben Teixeira [1], João Barreiros Rodrigues [2,*], Jaime Aguiar [1,2], Jiao Li [3] and José T. de Sousa [1,2,*]

[1] IObundle, Lda., 1000-029 Lisbon, Portugal; ruben.teixeira@tecnico.ulisboa.pt (R.T.); jaime.a.aguiar@tecnico.ulisboa.pt (J.A.)
[2] INESC-ID, INESC INOV, Instituto Superior Técnico, Universidade de Lisboa, 1000-029 Lisbon, Portugal
[3] Microelectronics Research Development Center, School of Mechatronic Engineering and Automation, Shanghai University, Shanghai 200444, China; lijiaoshu@shu.edu.cn
* Correspondence: joao.b.rodrigues@inov.pt (J.B.R.); jose.t.de.sousa@tecnico.ulisboa.pt (J.T.d.S.)

Abstract

Edge inference on resource-constrained embedded nodes demands accelerators that are energy-efficient and compact. This paper presents Versat-AI, an open-source compiler that accepts a standard Open Neural Network Exchange (ONNX) model and generates a complete, synthesisable RISC-V System-on-Chip (SoC) with an embedded CGRA accelerator. The key innovation is applying a known hardware merge strategy to collapse structurally compatible neural network operators into a physical CGRA instance. The Versat-AI compiler also derives memory-mapped interconnects, firmware drivers, and RISC-V application software co-generated by the Py2HWSW SoC framework, eliminating the manual hardware/software co-design effort that previously tied this accelerator's own design lineage to a single target network. The next phase of the project is to extend this same automatic derivation from sizing the operator vocabulary to sizing per-operator parallel instancing and bandwidth, following the bandwidth-matched scaling principle already demonstrated, by hand, in this accelerator's own design lineage. The current phase of the project has succeeded in creating a sound automation flow that produces an accelerator that maps each operator onto a single physical datapath instance and occupies 8763 LUTs, 9833 flip-flops, 4 DSPs, and 202 BRAMs on a Xilinx Kintex UltraScale field-programmable gate array (FPGA)—a footprint unchanged across all evaluated models regardless of size—and draws 0.65 W (1.96 W for the complete SoC including the DDR4 controller, by Vivado post-implementation power estimation), achieving 2.3× to 9× speedup over the software-only baseline produced by the same flow on four MLPerf Tiny benchmark tasks. The paper further examines the design choices that delimit this first phase—single-precision arithmetic, a single datapath instance per operator, and the block-RAM cost of the accelerator's streaming buffers—and sets out the path to quantised integer support and parallel operator instancing. A condensed account of the two-decade lineage of reconfigurable accelerators and open-source SoC platforms that motivated Versat-AI's design is also given.

Keywords: edge inference; CGRA; ONNX; RISC-V SoC; FPGA; hardware accelerator; deep learning; hardware/software co-generation; neural network; open-source; low power; energy efficiency

Check for updates

Academic Editor: Mario Casu

Received: 15 July 2026
Revised: 19 August 2026
Accepted: 25 August 2026
Published: 4 September 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

The proliferation of deep learning has created demand for inference at the edge: smart sensors, industrial monitors, and personal devices must classify, detect, or predict in real time within strict power and area budgets. Fixed-function Application Specific Integrated Circuits (ASICs) satisfy those budgets but cannot adapt when models evolve; cloud offload reintroduces latency and connectivity dependence; Graphics Processing Units (GPUs) and high-end processors offer flexibility at power levels embedded nodes cannot sustain [1]. Coarse-Grain Reconfigurable Arrays (CGRAs) occupy the promising middle ground: word-level reconfigurability keeps area and energy close to ASIC levels while preserving post-fabrication programmability [2].

Three problems, however, have limited CGRA adoption in edge Artificial Intelligence (AI) deployment. First, the *compilation problem*: identifying and mapping kernels onto CGRA fabric—effectively High-Level Synthesis (HLS) for CGRAs—has resisted full automation for two decades; existing tools require expert intervention, do not accept high-level model descriptions as input, or both. Second, the *area scaling problem*: many CGRA-based accelerator designs sidestep temporal reuse by replicating a physical structure more than necessary, leaving the resulting design unsuitable for resource-constrained targets where silicon area is a hard constraint.

A complementary obstacle is the *SoC integration problem*: deploying any accelerator in a production-quality embedded system requires not only the accelerator Register Transfer Level (RTL) but also memory interfaces, bus wrappers, peripheral drivers, and firmware. Assembling such a system using standard tools has historically proved far harder than designing the accelerator itself and has discouraged the use of CGRAs in production systems.

Versat-AI addresses all three challenges. It accepts a standard Open Neural Network Exchange (ONNX) [3] model as input and automatically produces a complete, synthesisable RISC-V System-on-Chip (SoC) with an embedded CGRA accelerator. The accelerator is generated by applying the hardware merge strategy of [4] to ONNX operator mapping: structurally compatible operators are collapsed into a CGRA instance, so the accelerator's hardware directly reflects the operator vocabulary the input model actually uses—rather than a designer-fixed, model-independent template or a hardware pipeline allocated per layer—keeping footprint nearly independent of model size. System integration is handled by the Py2HWSW [5] SoC generator, which produces memory-mapped interconnects, Control and Status Register (CSR) drivers, and firmware from the same description. The contributions of this work are:

- **Hardware merge strategy for ONNX operators.** The unit merge and reuse strategy of [4]—previously applied to DSP coprocessors—is applied here for the first time to neural network operator mapping. Structurally compatible operators are collapsed into CGRA configurations, producing an accelerator whose area is nearly independent of model size.
- **Automated Model-to-Runtime flow.** Versat-AI co-generates the hardware accelerator, memory-mapped interconnects, firmware drivers, and Reduced Instruction Set Computer V (RISC-V) application software from a single ONNX input, integrating with the Py2HWSW [5] SoC generator for a fully automated path from model to a running Field Programmable Gate Array (FPGA) system.
- **MLPerf Tiny evaluation.** The system is validated on four MLPerf Tiny inference tasks [6], achieving 2.3× to 9× speedup over the software-only baseline generated by the same flow on a Xilinx Kintex UltraScale FPGA, with a single automatically-derived datapath instance per operator—evidence that the automation itself is sound, ahead of extending it to automatically size per-operator parallel instancing and bandwidth.

https://doi.org/10.3390/jlpea16030036

- **Architectural lineage and SoC integration context.** The background section summarises the evolution of the reconfigurable accelerators and open-source SoC platforms that shaped Versat-AI's design, and identifies which of their components Versat-AI inherits.

The merge strategy at the core of this work is itself a known technique, and the accelerator it produces is built on a substrate developed over two decades. Table 1 therefore records, component by component, what is newly developed here, what is adapted, and what is reused unchanged.

Table 1. Separation of new, adapted, and reused components in Versat-AI.

Component	Status	Origin
Merge of ONNX operator datapaths	New	Section 4.1.3
Inter-configuration delay correction	New	Section 4.1.3
Versat Spec operator library	New	Section 4.1.1
ONNX parser, scheduler, allocator	New	Section 4.2
Model-to-runtime integration flow	New	Section 4.3.2
Compatibility-graph unit merge	Adapted	Section 4.1.3, [4,7]
Graph-level format optimisations	Adapted	Section 4.2
Bandwidth-matched sizing discipline	Adapted	Section 3.1, [8]
Versat CGRA and self-reconfiguration	Reused	[9,10]
VRead/VWrite streaming units	Reused	[8]
Py2HWSW, IOb-SoC, VexRiscv	Reused	[5,11]

2. Related Work

2.1. CGRAs and Programmable Accelerators

Coarse-grained reconfigurable arrays sit between FPGAs and fixed-function ASICs in the programmability–efficiency space: word-level reconfigurability keeps area and power close to ASIC levels while preserving post-fabrication flexibility [2]. Early architectures such as RaPiD [12] and ADRES [13] established the canonical template of interconnected processing elements with local memories, but both relied on offline loop-kernel mapping tools that never achieved the automation level of FPGA place-and-route, limiting practical adoption. That tooling gap has driven sustained research effort: Morpher [14] integrates CGRA design, compilation, and simulation into a single open-source framework targeting edge AI kernels, while RipTide [15] and Snafu [16] compile arbitrary C programs into energy-minimal spatial dataflow graphs, demonstrating that near-ASIC efficiency is achievable with the right compiler substrate. OpenEdgeCGRA [17] contributes an open-hardware CGRA specifically evaluated on convolutional layers at the edge, providing a useful area and throughput baseline for CNN-targeted designs.

Despite this progress, all of these approaches require an explicit kernel mapping step—either offline scheduling or C-level entry—and none provides a direct path from an ONNX model to a synthesisable CGRA SoC: even where a compiler front-end exists, its output is the CGRA accelerator or its configuration alone, not the surrounding interconnects, firmware, and application software a deployable SoC requires. Versat addresses the mapping difficulty through self-reconfiguration: the RISC-V host programs the array at runtime from software, replacing the offline spatial mapper with a conventional instruction stream. The compiler layer above it—Versat-AI—further applies the hardware merge strategy of [4] to collapse structurally compatible ONNX operators into CGRA configurations.

2.2. Edge AI SoC Platforms

The dominant open-source approach to edge AI SoC design is the Parallel Ultra-Low-Power (PULP) platform [18], whose commercial derivative GAP8 pairs a single-core controller domain with a cluster of eight RISC-V cores and achieves competitive energy efficiency on quantised inference workloads. The PULP-NN library [19] provides hand-tuned quantised inference kernels for this multi-core substrate, covering common Convolutional Neural Network (CNN) and Digital Signal Processor (DSP) primitives. The successor GAP9 extends the cluster to nine cores, adds floating-point support across all cores, and targets 50 GOPS at under 50 mW on a 22 nm process—making it the principal industrial benchmark for ultra-low-power edge inference. These platforms demonstrate that a tightly-coupled RISC-V cluster paired with a shared Tightly Coupled Data Memory (TCDM) is highly effective for data-parallel workloads, but inference relies on manually written, architecture-specific libraries rather than an automated model-to-hardware flow.

On the open-source SoC platform side, X-HEEP [20] provides a configurable and fully open-source RISC-V microcontroller with a standardised accelerator interface (XAIF) that has been taped out in 65 nm CMOS, making it the closest architectural cousin to IOb-SoC in terms of openness and configurability, though it does not itself generate an accelerator from a model. Gemmini [21], integrated into the Chipyard [22] SoC generator ecosystem, attaches a parametric systolic array to a Rocket RISC-V core via the RoCC interface and provides a full quantised execution flow from ONNX, but the array's dimensions are a design-time choice fixed independently of any particular model. CGRA4ML [23] goes further still, producing a complete, taped-out SoC—RTL, AXI Direct Memory Access (DMA) interfaces, firmware, and Arm integration— built around a CGRA that shares resources across network layers through runtime-multiplexed processing elements; however, its array dimensions are likewise a fixed, user-specified template, and its front end accepts only QKeras-trained models rather than a standard interchange format. None of these three, unlike Versat-AI, derives the accelerator's hardware configuration automatically from the operator vocabulary of an arbitrary ONNX model.

2.3. Inference Compiler Frameworks

At the software compilation end, Apache TVM [24] and ONNX Runtime lower ONNX graphs into optimised kernels for existing processors and accelerators without generating hardware. FINN [25] and hls4ml [26] take the next step by synthesising model-specific hardware: FINN produces streaming pipelines of processing elements matched to the network topology for FPGAs, while hls4ml generates HLS blocks for low-latency scientific applications. Both are mature, widely used frameworks, but the hardware they produce scales in resource usage with model depth and width, making them unsuitable for area-constrained targets where the FPGA fabric is a hard limit. CONVOLVE [27] addresses CGRA targets specifically, lowering ONNX through an MLIR pipeline to an OpenASIP-based CGRA back-end with automated kernel scheduling. Vitis AI [28] provides a complete deployment flow for AMD FPGA devices using the Deep-learning Processing Unit (DPU) overlay but targets mid-to-high-density FPGAs and is not open-source below the library level. VTA [29] follows a similar template pattern on the hardware side: a configurable but design-time-fixed tensor accelerator driven by a TVM-hosted just-in-time compiler, so, as with Gemmini and CGRA4ML above, the array itself is chosen once and reused across models rather than reshaped per operator vocabulary. NNgen [30] instead synthesises a model-specific accelerator directly from a trained network description through an HLS-style back-end, placing it alongside FINN and hls4ml: its hardware footprint scales with the model rather than remaining fixed. NVDLA [31], like Vitis AI's DPU, pairs a configurable

but still template-based convolution engine with a compiler that maps an input graph onto it, rather than deriving the datapath itself from the model's operator set.

Versat-AI occupies a distinct position in this landscape: it applies the hardware merge strategy of [4] to ONNX operator mapping, identifying structurally compatible operators across different graph nodes and collapsing them into CGRA configurations. The result is a fixed-footprint accelerator whose area is independent of model size, integrated with the Py2HWSW [5] SoC generator to produce a complete synthesisable RISC-V SoC—including memory-mapped interconnects, peripheral wrappers, and firmware drivers—from a single ONNX input.

3. Background

3.1. Reconfigurable Accelerator Architecture

Versat-AI draws its core architectural ideas from a line of reconfigurable accelerators developed over two decades, each resolving a limitation of its predecessor. The thread runs from SideWorks (hardware merge for DSP coprocessors) through Versat (generalised mini-CGRA with full crossbar) and DeepVersat (pipelined deep-learning array) to the YOLO accelerator, where the bandwidth-disciplined design philosophy and the VRead/VWrite memory units that Versat-AI inherits directly were first introduced.

3.1.1. SideWorks and Versat

SideWorks [32] is a reconfigurable DSP coprocessor in which structurally compatible operators from different algorithm modules are merged into a single shared physical instance via a partial read/write crossbar, so that silicon footprint is determined by the operator vocabulary rather than the number of supported algorithms [4]. Its key mechanisms are a configurable multi-level Address Generation Unit (AGU) that reproduces any stride-based access pattern through reconfiguration alone, a programmable timing unit that compensates automatically for unequal pipeline latencies, and partial reconfiguration at individual register granularity, so that switching operators requires only a small number of CSR writes. BDTI independently validated the approach in 2009, the SideWorks-based CWcomp4465 core achieving the smallest footprint in its benchmark comparison by more than a factor of two [33]. This hardware merge strategy is the direct conceptual precursor to Versat-AI, where it is applied for the first time to ONNX neural network operators.

Versat [9,10] generalised the merge concept into a domain-independent mini-CGRA by replacing SideWorks's fixed operator vocabulary with an implicit, full crossbar interconnect among 15 heterogeneous functional units, as illustrated in Figure 1, eliminating the placement-and-routing step that dominates mapping tool complexity in conventional 2D-mesh CGRAs [2] at only 4% area overhead. This full-crossbar generality comes at a routing cost that grows quadratically with the number of functional units: Versat's footprint is fixed for its 15-unit vocabulary, but the same approach does not scale to a larger functional-unit set without the interconnect itself becoming the dominant, unroutable cost. A 16-instruction controller generates configuration sequences at runtime from compact parametric routines (*self-reconfiguration*), with a shadow register hiding reconfiguration latency behind the current execution phase—the mechanism Versat-AI inherits for its own operator switching. DeepVersat [34] subsequently extended Versat into a linear pipeline of identical stages to meet the compute density of deep convolutional networks, mapping one network layer per hardware stage and forwarding intermediate feature maps directly between stages without external memory round-trips, thereby scaling compute capacity through local stage-to-stage connections rather than through an interconnect whose routing cost would explode with functional-unit count. Its reuse, however, is by *repetition*: each stage is a homogeneous copy of the same elemental Versat block, and the pipeline's

depth and topology are fixed at synthesis time to one specific network. This motivated the compiler-driven merge strategy at the core of Versat-AI, which instead aggregates *heterogeneous* functional units into a single shared structure and reuses the routing itself across operator configurations, keeping interconnect limited to what is strictly necessary rather than a general crossbar or a chain of identical copies.

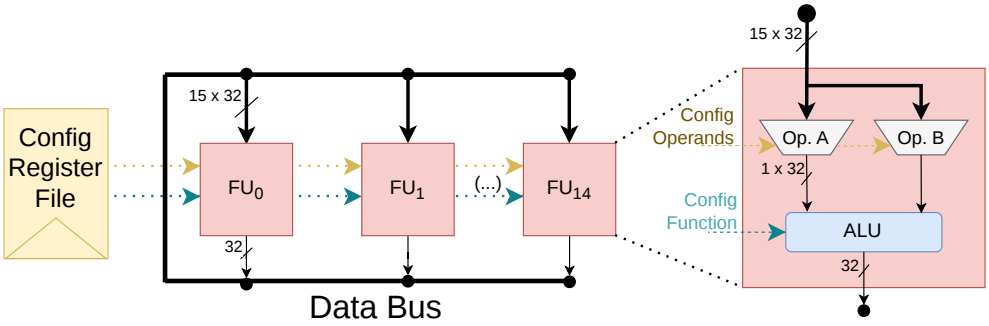


Figure 1. Original Versat's data engine structure: 15 functional units (FU_0–FU_{14}) share a data bus (black) and are each configured by the Config Register File along the operand-select path (gold, dashed) and the function-select path (teal, dashed); "(...)" marks the elided units FU_2–FU_{13}. The right-hand detail shows a single unit: two operand multiplexers (Op. A, Op. B) feeding an ALU.

3.1.2. Versat for Object Detection

Rather than extending DeepVersat, a different path was taken in [8,35]: a hand-crafted accelerator designed from first principles around the bandwidth constraints of Tiny-YOLO inference, integrated into a complete RISC-V SoC on a Kintex UltraScale KU040 FPGA. A topview of the YOLO Versat architecture is illustrated in Figure 2. The work established two principles and one hardware abstraction that Versat-AI inherits directly.

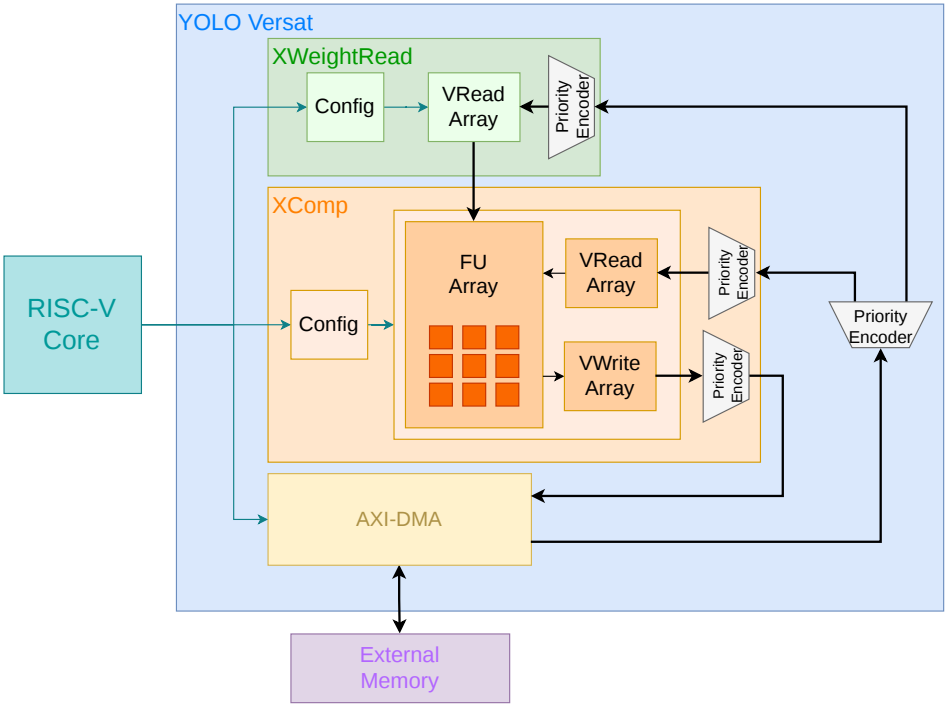


Figure 2. YOLO Versat architecture topview.

The first principle is that memory bandwidth, not arithmetic throughput, is the binding constraint in CNN inference [1]: adding Multiply And Accumulate (MAC) units beyond the point the memory bus can sustain yields no throughput gain. The design therefore sizes compute to match the available bus, resulting in 832 (YOLOv3-Tiny) or 1248 (YOLOv4-Tiny)

MAC units fed by a 256-bit DMA interface and ensures that the bus is kept fully utilised at all times. Versat-AI carries this discipline forward: the current implementation instantiates ONNX operators of different types and makes sure the units are never data-starved.

The second principle is full pipeline coverage: unlike prior work that accelerated only the CNN backbone, the YOLO accelerator handles the complete inference pipeline—pre-processing, all CNN layers, and post-processing—without returning control to the Central Processing Unit (CPU) between steps [8]. Versat-AI adopts the same goal: the ONNX graph is mapped end-to-end onto the accelerator, with the RISC-V CPU managing only scheduling and I/O.

The hardware abstraction is the VRead/VWrite pair. **VRead** is a dual-port embedded memory: one port is driven by an AXI-DMA engine that fetches data from external memory in configurable AXI4 bursts over the 256-bit bus, while the other port is driven by an independent AGU that sequences the buffered words into the arithmetic datapath. **VWrite** is the symmetric counterpart: an internal AGU writes computation results into one port while an external AGU simultaneously drains the other port back to external memory via DMA. Both units operate as ping-pong buffers, sustaining a continuous data stream [8]. Versat-AI inherits both units unchanged and shares them across every supported ONNX operator.

The YOLO accelerator's area footprint (103–147 k LUTs, 832–1248 DSPs on the KU040) is itself constant with respect to the network: its fixed, bandwidth-matched functional-unit count is reused across the entire inference pipeline, the same constant-footprint outcome DeepVersat's stage-chaining and Versat-AI's compiled merge each reach by different means. What it lacks is generality: reaching that reused, bandwidth-matched datapath required a designer to hand-derive it for Tiny-YOLO specifically, so supporting a different network or operator set means repeating that manual design effort from scratch. Versat-AI resolves this by compiling the same reuse strategy automatically for an arbitrary ONNX model: its merge algorithm aggregates the heterogeneous functional units the target operator set requires and reuses both the units and their strictly-necessary routing across operators, producing an accelerator of under 9 k Look-Up Tables (LUTs) regardless of model size without any per-network manual redesign, at the cost of peak throughput that future parallel operator instances will recover.

3.2. Open-Source SoC Platform

Assembling a production-quality SoC from open-source components is a recurring challenge for researchers and educators: the ecosystem offers processors, bus standards, simulation tools, and FPGA support packages, but composing them into a reliable, maintainable platform is rarely as straightforward as the individual tools suggest. Integration failures are common yet almost never published, leaving each new group to rediscover the same obstacles independently. This section documents that experience and the purpose-built platform that emerged from it.

3.2.1. IOb-SoC

IOb-SoC is the open-source RISC-V SoC template that emerged from two prior integration attempts. Blackbird [36] assembled an SoC around the OpenRISC 1000 processor [37], the ORPSoC generator [38], and a Wishbone bus [39] but stalled on the immaturity of the OpenRISC ecosystem and on hard dependencies on specific FPGA boards. Warpbird [40] replaced OpenRISC with a 64-bit RISC-V Rocket core [41] generated in Chisel [42]/Scala and migrated the bus to AXI4, but the generator proved too monolithic to decompose and adapt, and its debug subsystem could not be integrated into the team's verification flow [43]. Both attempts nonetheless established the unified verification strategy the platform still uses—RTL simulation with Icarus Verilog [44], cycle-accurate simulation via Verilator [45],

and FPGA emulation, all driven from the same test program—and led to the conclusion that a purpose-built, modular SoC template was the only viable path. It is built around a softcore CPU (such as VexRiscv [11] or Ibex [46]) and uses an AXI4 master bus for external memory access, connecting to either on-chip block RAM or a third-party DDR controller. A prebuilt peripheral library provides bare-metal drivers for standard components such as UART and timer.

IOb-SoC is now described in Python 3.12 and generated by the Py2HWSW framework [5], which automates the production of the Verilog sources, bus wrappers, and software drivers for every peripheral or accelerator added to the system.

The framework supports RTL simulation with Icarus [44], Verilator [45], Xcelium (Cadence, including code coverage), Questa (Siemens), and VCS (Synopsys); FPGA synthesis with Vivado (AMD) and Quartus (Altera); ASIC synthesis with Genus (Cadence); and static linting and CDC/RDC checks with ALINT-PRO (Aldec) and SpyGlass (Synopsys), making the same SoC description target-agnostic across the full spectrum from open-source simulation to commercial ASIC sign-off.

3.2.2. Py2HWSW

Py2HWSW [5] is an open-source Python-based framework that acts as an integrator and generator of modular system-on-chip platforms built around RISC-V processors with instantiated peripherals.

System specifications are expressed in Python using high-level abstractions describing processor configuration, memory mapping, and peripheral connectivity. From this description, Py2HWSW automatically generates synthesizable Verilog for system components, memory-mapped interconnect logic, peripheral wrappers, and software support files.

A key capability of Py2HWSW is the integration of heterogeneous components into a coherent system. It connects RISC-V cores with standard peripherals (e.g., UART, timer) and custom hardware accelerators within a unified memory-mapped architecture, enabling seamless interaction between processor software and accelerators such as those compiled by Versat.

4. Versat-AI

Versat-AI [47] is a hardware-accelerated inference runtime generator for ONNX models that functions as a Model-to-Runtime compiler. It automatically generates both the specialised hardware accelerator and the embedded software required to execute deep learning inference on a System-on-Chip (SoC). Unlike traditional CGRA-based approaches that focus solely on operator mapping, Versat-AI deploys a complete hardware/software system and its verification environment, as shown in Figure 3.

As shown in Figure 3, three tools cooperate to produce the final system. The Versat Compiler takes the Versat Spec file together with the hardware description of each functional unit and synthesises the merged CGRA accelerator. The ONNX Parser takes only the target ONNX model and produces the compiled inference program and its supporting data artifacts. Py2HWSW then assembles the complete system, combining the artifacts generated by the Versat Compiler and the ONNX Parser with the `VersatAI.py` system description and the `VersatAI.c` static inference firmware. During build, Py2HWSW orchestrates system generation, integrating the Versat accelerator with the VexRiscv [11] CPU, memory subsystem, and peripherals to produce the final hardware/software system and test environment.

4.1. Versat Compiler

The Versat Compiler, whose workflow is illustrated in Figure 4, takes two inputs: the Verilog hardware description of each functional unit (e.g., integer adder, floating-point multiplier) and a Versat Spec file, describing a hardware accelerator for each neural network operator in terms of those units. The Unit Registry parses the Verilog units, reading off each one's port list and the latency attribute annotated on its interface, and registers it as a reusable elemental building block. The Specification Parser reads the Versat Spec file and, for each operator module, builds a graph representation of its datapath and configuration logic over the units supplied by the Unit Registry. The Versat Specification and Hardware units database are detailed in Section 4.1.1. The per-operator graphs are then flattened to their elemental units, so they can be shared among operators, and merged into a single graph representation describing the Versat Accelerator datapath and configurability, from which the Output Stage generates the CGRA Accelerator description in Verilog and its respective driver in C. The merge strategy is described in depth in Section 4.1.3.

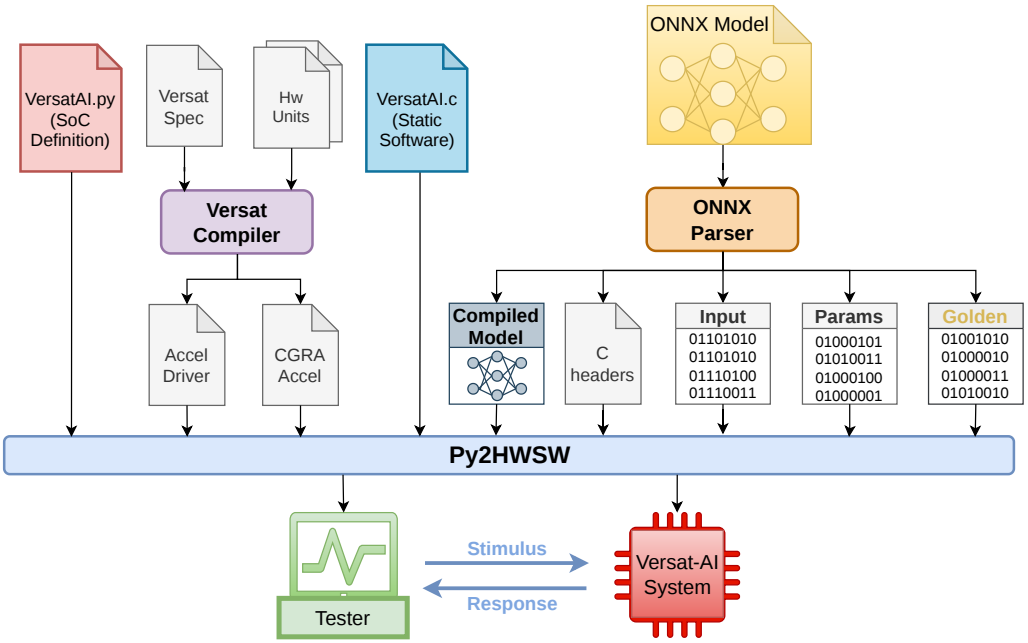


Figure 3. Versat-AI workflow.

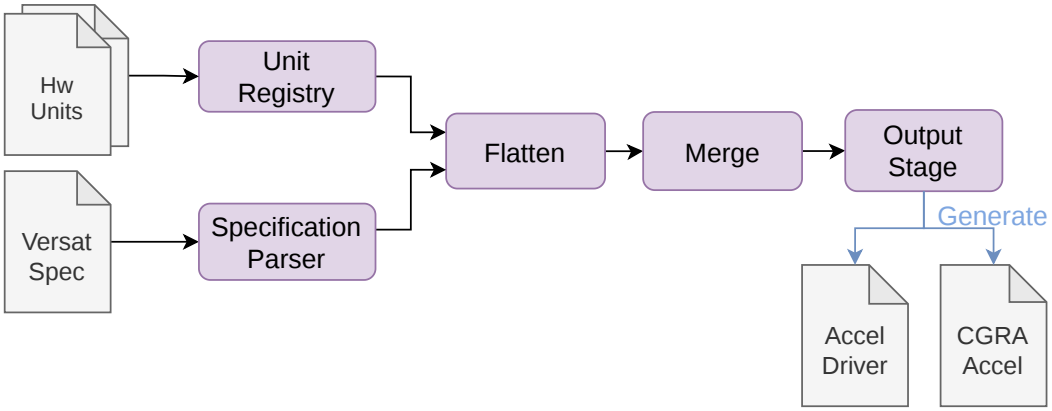


Figure 4. Versat compiler workflow.

4.1.1. Versat Specification and Hardware Units

The Versat specification contains the description of the operators to be synthesised, working as a higher-level representation of the functionalities to be implemented by the

CGRA. Each operator module has a datapath description in an HDL-like syntax and multiple configuration descriptions in a C-like syntax.

The functional units instantiated by these operator datapaths, and their respective pipeline depths, are described in Table 2. Each unit exposes its own pipeline latency to the compiler via a `versat_latency` Verilog attribute on its output; the compiler uses these per-unit values to statically schedule the merged CGRA datapath and to derive the end-to-end operator depths.

Table 2. Functional unit description and latency.

Unit	Description	Depth (Cycles)
VRead	Streams operands from local memory into the datapath under a configurable access pattern (linear, strided, broadcast, 2-D windowed)	3
F_Mul	Single-precision floating-point multiplier	4
F_Add	Single-precision floating-point adder	5
F_Accum	Single-precision floating-point accumulator; sums a stream of partial results over a configurable window	4
F_AccumMax	Streaming maximum-reduction unit; compares each incoming sample against a running maximum over a configurable stride	1
Relu	Rectified-linear-unit activation	1
Const	Drives a fixed, per-configuration constant value onto the datapath (e.g., an accumulator seed or scale/bias operand); no data-dependent computation, so it adds no pipeline latency	0
VWrite	Streams datapath results back to local memory under a configurable access pattern; terminal stage of the pipeline	NA

The seven operators accelerated by the Versat-AI CGRA are described in Table 3. The table lists, for each operator, its name, the principal hardware units its datapath instantiates, and its latency in clock cycles. Because these datapaths are graphs rather than simple chains—units can fan out to, or be shared between, multiple downstream units—the list of units is given as a set rather than as a connection sequence; Section 4.1.3 details how the units are actually wired and subsequently shared across operators. These seven are the operators with a CGRA datapath; as described in Section 4.2, the flow additionally accepts ONNX operators that have a software-only implementation and are dispatched to the RISC-V CPU, so the set of operators the flow can execute is larger than the set it accelerates.

Table 3. Supported ONNX operators and hardware mapping.

Operator	Units	Depth (Cycles)
Add	VRead, F_Add, VWrite	8
MatMul	VRead, F_Mul, F_Accum, VWrite	11
Conv	VRead, F_Mul, F_Add, F_Accum, Const, VWrite	17
AveragePool	VRead, F_Accum, F_Mul, VWrite	11
MaxPool	VRead, F_AccumMax, VWrite	4
BatchNorm	VRead, F_Mul, F_Add, VWrite	12
ReLU	VRead, Relu, VWrite	4

4.1.2. The MatMul Operator

This section uses the `MatMul` operator as a running example to illustrate the Versat Spec syntax; its specification is given in Listing 1. Given a matrix multiplication $C = A \times B$, where A is $N \times K$ and B (pre-transposed) is $M \times K$, one call to the module computes a single row of the $N \times M$ result matrix: it takes the dimensions K and M, a pointer to the corresponding row of A, and a pointer to the transposed matrix B, and must be invoked N times, once per row of C, to produce the complete result.

Listing 1. MatMul operator description in the Versat Spec.

```
// Computes one row of C[N*M] = A[N*K] * B[K+M]
module MatMul(){
    VRead ARow;
    VRead BColT;
    F_Mul mat_mpy;
    F_Accum Accum;
    VWrite C;

    ARow -> mat_mpy:0;
    BColT -> mat_mpy:1;
    mat_mpy -> Accum;
    Accum -> C;

    config All(ARow_buf:Buffer, BColT_buf:Buffer, C_buf:Buffer, M:
        Dyn, K){
        for y 0..M{
            for x 0..K {
                ARow = ARow_buf[x];
                BColT = BColT_buf[y * K + x];
            }
        }
        Accum.strideMinusOne = K - 1;
        for x 0..M {
            C_buf[x / K] = C;
        }
    }
}
```

A module is declared with the `module` keyword, followed by its unit declarations (`Type name;`, e.g., `VRead ARow;`) and a connection graph describing how those units' ports are wired together. Each connection is written `source -> destination`; when no port is given, port 0 is assumed on both ends, and `unit:X` selects port `X` explicitly (so `Accum -> C` is shorthand for `Accum:0 -> C:0`).

A graphical representation of the `MatMul` module is given in Figure 5. This figure also contains the accumulated latency after each unit. The VRead units ARow and BcolT introduce a three-cycle delay once the accelerator is started. The latency monotonically increases as we go down the Directed Acyclic Graph (DAG) and reaches a maximum at the terminal C VWrite node, from where it is written to memory.

After the connections, one or more `config` functions describe how the module is parameterised at runtime. Versat spec files support three function kinds—`config`, `mem`, and `state`—for writing unit configuration, accessing memory-mapped units, and reading unit state back, respectively; only `config` is needed here. Each `config` function is named (the name becomes the generated C function that the firmware calls, e.g., `All` in the listing) and takes parameters of the form `name:modifier`. In the example, `:Buffer` marks a pointer argument, while integer arguments are either fixed (e.g., K) or dynamic `:Dyn` (e.g., M). Fixed integer parameters are not checked for hardware size compatibility, whereas dynamic parameters inform the compiler that the hardware size may be exceeded, and the compiler

proceeds to automatically fold the loop the number of times necessary to complete the loop with the existing hardware. The distinction between fixed and dynamic parameters is not strictly necessary and the compiler can be upgraded to dispense with it.

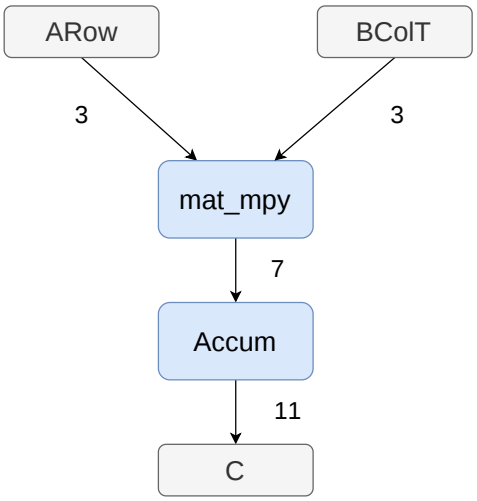


Figure 5. Graph corresponding to the MatMul module specification.

The assignment `BColT = BColT_buf[y * K + x]` inside the nested `for y` loop is not unrolled into literal per-cycle values: it symbolically describes the address pattern that `BColT`'s underlying `VRead` unit should generate over time—at loop iteration `(x,y)`, the unit is to produce the value found at `BColT_buf[y * K + x]`, a linear address expression over the loop indices. `ARow_buf` and `BColT_buf` are addresses in external Double Data Rate (DDR) memory, while the VRead and VWrite units each hold a small internal on-chip buffer that is filled from (or drained to) DDR sequentially; the address expression given in the `config` function addresses that fast internal buffer to produce (or consume) one value per cycle on the datapath side. This split exists because DDR only supports efficient sequential/burst access, whereas the arbitrary access patterns Versat operators need (linear, strided, broadcast, 2-D windowed, per Table 2) are only practical against the fast internal buffer. `Accum.strideMinusOne = K - 1;` sets a config wire directly: `F_Accum` counts accumulated values from 0, so `strideMinusOne` is the zero-based terminal count at which it flushes its sum, precomputed here to avoid a hardware subtractor. Finally, `C_buf[x / K] = C` demonstrates a non-linear address expression on the write side: integer division collapses each group of `K` consecutive cycles onto the same internal-buffer address, so only the last (fully accumulated) value of each group is the one that persists once that address is eventually drained to `C_buf` in DDR.

4.1.3. Merge Algorithm

Each operator is specified independently in the Versat Spec, following the structure illustrated for `MatMul` in Section 4.1.2. As noted in Section 4.1.1, functional units can be shared across such operators; this section describes the merge algorithm that performs that sharing.

A `merge` statement in the spec file instructs the compiler to fuse the specified accelerators into a single CGRA. The compiler resolves the merge with a compatibility-graph formulation in the tradition of prior datapath-merging techniques [7,48]: structurally compatible hardware units across modules are identified and overlapped into CGRA configurations, trading off interconnect silicon area for a more complex compiler that can reuse the same hardware units across different operators. Within the Versat line of work specifically, this same reuse strategy was first applied to DSP coprocessors [4]; here we

extend it to the novel context of neural network operator mapping, and it differs from the original Versat CGRA [10], which used a full crossbar to connect all units.

To build the CGRA, each input operator is first flattened into a graph of its functional units, and the merged graph is grown incrementally: it is initialised as a copy of the first operator, and each remaining accelerator is folded in one at a time. For every pair of units between the incoming operator and the merged graph so far, structural compatibility is evaluated and recorded as a node of a different graph called the compatibility graph; two candidate mappings are connected by an edge only if they can be adopted together, since merging one pair of units can preclude other pairings.

Figure 6 illustrates this process for two example operators, X and Y. X chains `VRead(A)`, `Relu`, `Float_Add` along one input path and `VRead(B)` directly into `Float_Add` along the other; Y instead has a single `VRead(A)` whose output diverges into two paths, one through `Relu` and one without, that reconverge at `Float_Mul`. Both write their result through a `VWrite`. Comparing the two unit-by-unit gives the compatibility graph's nodes: each operator has exactly one `Relu` and one `VWrite`, so each admits only a single candidate mapping, whereas X's two `VRead` instances can each map to Y's single `VRead(A)`, giving two candidate unit mappings. Of the connections in the two datapaths, only the `VRead(A) -> Relu` edge exists in both operators, so it is the sole candidate edge mapping.

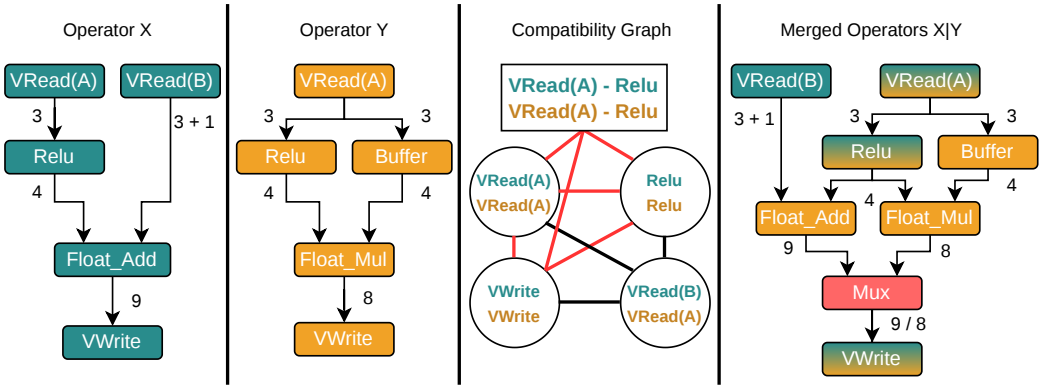


Figure 6. Example of two operators, their compatibility graph, and the merged result. Operator X nodes are shown in green and Operator Y nodes in orange; in the compatibility graph, circles are candidate unit mappings (X node over Y node) and the rectangle is the candidate edge mapping, with the maximum clique highlighted in red. Numbers on edges are the accumulated latency in cycles.

In the compatibility graph, circles denote candidate unit mappings and the rectangle denotes the candidate edge mapping just described; an edge between two candidates indicates they can be adopted together without structural conflict, and the maximum clique—highlighted in red—gives the largest consistent set of shared mappings, with node colour coded by the input graph it originates from. For X and Y, this clique shares three units: their respective `VRead(A)` instances collapse into one, as do their `Relu` and `VWrite` instances, leaving X's `VRead(B)` as its own unshared instance. This is also why `VRead(A)` is shared over `VRead(B)`: sharing either is structurally possible, but sharing `VRead(A)` avoids inserting a multiplexer, since the `VRead(A) -> Relu` connection already existed as a candidate edge mapping. As with any port left with inputs from more than one configuration, a multiplexer is inserted to select between them: here, to route the output of `Float_Add` or `Float_Mul` into the shared VWrite unit depending on whether X or Y is the currently configured operator. Each configuration's data path is wired to the multiplexer port matching the index of its input operator, an association recorded in the mapping described above so it can be exported as firmware configuration data; the firmware then drives the multiplexer's select input at runtime.

In general, finding the largest set of mutually compatible mappings is a maximum clique [49,50] problem on the compatibility graph, computed as described in Listing 2, so that as many units as possible are safely collapsed into shared instances; units without a compatible match are instantiated anew. This search uses a branch-and-bound algorithm [51], giving it a worst-case complexity of $O(3^{M/3})$ [52] for a compatibility graph with M vertices, bounded by a configurable timeout (10 s by default) to avoid hanging on large or complex consolidation graphs. This bound is deliberately conservative: for the compatibility graphs arising from our current operator vocabulary (Table 3) the search completes in milliseconds and the timeout never triggers in practice; we expect it to remain comfortably below the default even for substantially larger graphs (on the order of 30–40 vertices). Merging more than two operators is handled by repeating this fold-in step iteratively, one accelerator at a time.

Listing 2. Branch-and-bound time-limited MaxClique algorithm.

```
Clique MaxClique(Graph G, double timeLimit /* seconds, default 10.0
    */){
    Clique best = EmptyClique();
    double startTime = GetTime();
    for (Vertex v : G.vertices){
        VertexSet candidates = Neighbors(v);
        Search(v, candidates, 1, &best, startTime, timeLimit);
    }
    return best;
}

void Search(Vertex v,
            VertexSet candidates,
            int size,
            Clique* best,
            double startTime,
            double timeLimit){
    /* Timeout */
    if (GetTime() - startTime >= timeLimit)
        return;

    if (Empty(candidates)){
        if (size > Size(*best))
            UpdateBestClique(best);
        return;
    }

    while (!Empty(candidates)){
        Vertex u = FirstCandidate(candidates);
        /* Branch-and-bound */
        if (size + Count(candidates) <= Size(*best))
            return;

        if (size + UpperBound(u) <= Size(*best))
            return;

        VertexSet next = candidates;
        Remove(next, u);
        next &= Neighbors(u);
        Search(u, next, size + 1, best, startTime, timeLimit);
        Remove(candidates, u);
    }
}
```

Merging units that were previously independent can introduce timing mismatches between paths that now feed a shared downstream unit: in the merged graph, edges are annotated with each path's global latency in cycles, and X's path through `VRead(A)` and `Relu` to `Float_Add` is longer than its direct path from `VRead(B)`. When the mismatched paths do not reconverge anywhere else before their common downstream unit—as with X's two independent inputs to `Float_Add`—Versat resolves the difference for free: because `VRead` and `VWrite` units have programmable address generators, the faster path's AGU is simply configured to start issuing (or accepting) addresses later, with no additional buffer hardware needed.

This AGU-offset trick does not work, however, when the mismatched paths reconverge from a single shared source: in Y, both branches leaving `VRead(A)` are driven by the same AGU, so they cannot be delayed independently before recombining at `Float_Mul`. In such reconvergent cases, a dedicated delay buffer is inserted into the faster (non-`Relu`) branch to re-synchronise the two operands; this buffer is visible in the merged graph in Figure 6 and kept there because X never required one.

Because a buffer added for one configuration can itself change the delay seen by others, the mappings and delay calculations are recomputed after every buffer insertion, repeating this correction until the timing of every configuration is consistent; this corrective flow is illustrated in Figure 7. This iteration is needed because delay buffers are runtime-programmable but still impose a hard floor of one cycle each, which is what triggers the loop: inserting a buffer to fix one configuration's timing can only add whole cycles, potentially unbalancing another configuration and forcing a further buffer elsewhere. Once this buffer-insertion process converges and the merged graph's topology is fixed, any additional delay beyond that first cycle is free—it is simply written into the corresponding operator's configuration, the same AGU-offset mechanism described above, with no further hardware changes required.

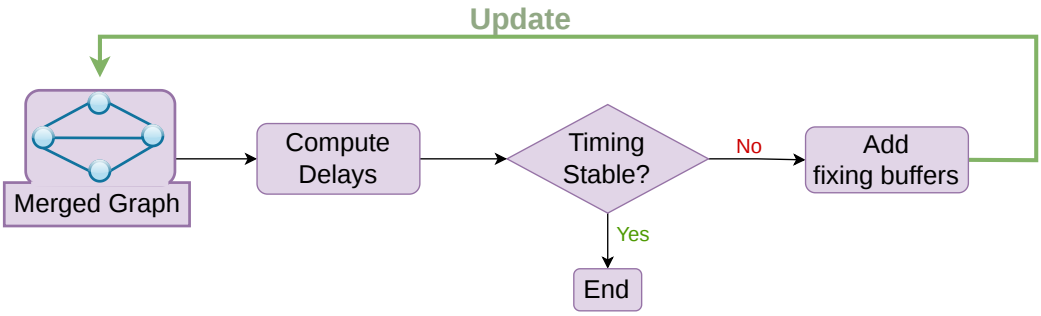


Figure 7. Delay correction in the merge algorithm.

Throughout the merge process, the compiler records, for each input operator, which merged-graph unit each of its original units was folded into, and the reverse: for each merged-graph unit, which original unit it stands in for under each operator. This mapping is what lets the compiler later recover, for any given operator, how the shared hardware must be configured to reproduce that operator's original dataflow.

With this merge strategy, the accelerator area is determined entirely by the number of unique hardware units required to support the full operator set, rather than the model size. Multiple operators of the same kind are planned, but their number will be tied to the parallelism opportunities, and not to the model size. Therefore, an example 50-layer ResNet uses the same silicon footprint as a five-layer one.

4.1.4. Dynamic Partial Reconfiguration

Because the merge algorithm collapses multiple operators onto shared hardware, switching between them at runtime requires reconfiguring the shared units' behaviour

without resynthesis. This is done through the Versat-AI CSR interface shown in Figure 8: the Config Reg/Config Shadow Reg datapath on the left, and the Read/Compute/Write pipeline schedule on the right.

At runtime, the firmware switches operators by writing to the configuration register accessed through this interface. Each functional unit's configuration is split into individually addressable fields, grouped by the pipeline phase they affect—Read, Compute, or Write—as detailed per unit in Table 4. A CSR write lands in the Config Reg and is then propagated, through a per-phase delay chain (no delay for Read, one register for Compute, two for Write—312, 1194, and 102 bits respectively, 1608 bits in total), into the Config Shadow Reg that actually drives the datapath. Because fields are individually addressable, switching to a new operator only requires rewriting the (typically small) subset of fields that actually differ from the current configuration, rather than the whole register file; Table 5 lists, for every supported operator, the hardware units it uses, the number of such fields that must be written, and their total bit width.

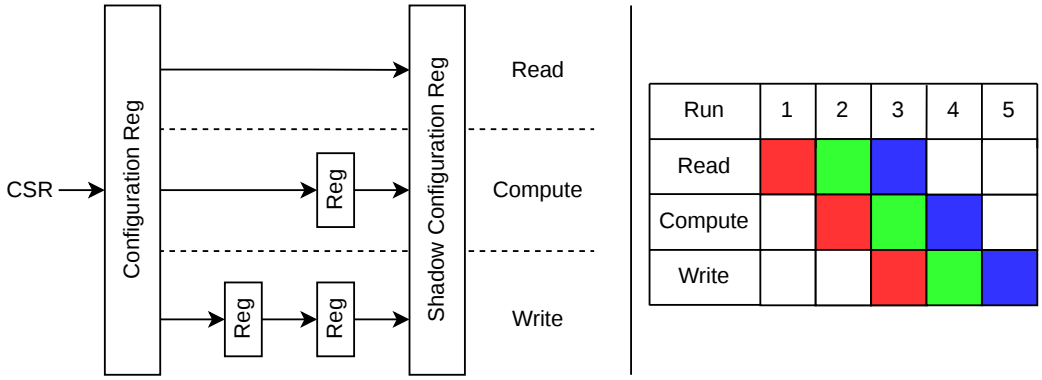


Figure 8. Versat-AI's CSR reconfiguration mechanism. Left: a CSR write lands in the Configuration Reg and reaches the Shadow Configuration Reg that drives the datapath through a per-phase delay chain—no delay for Read, one register for Compute, two for Write. Right: the Read/Compute/Write pipeline schedule, each colour a distinct run, with up to three runs in flight at once.

Table 4. Configuration cost per functional unit instance.

		Config. Bit Widths			
Unit	**Fields**	**Read**	**Compute**	**Write**	**Total**
VRead	23	104	265	0	369
VWrite	23	0	112	247	359
F_AccumMax	2	0	26	0	26
F_Accum	2	0	26	0	26
Const	1	0	32	0	32
Buffer	1	0	10	0	10
Mux	1	0	3	0	3

This makes reconfiguration both dynamic and highly partial: it happens at runtime, on demand, per operator switch, touching only the handful of fields that changed. Each field occupies its own 4-byte CSR address (Table 4), so writing one field costs exactly one RISC-V store-word instruction on the firmware's VexRiscv [11] core. Most fields configure a unit's address generator: `VRead/VWrite`'s AGU walks a nest of loops over the internal buffer, and each loop level needs its own start, increment, and stop values, with some levels additionally needing a duty-cycle count for strided or windowed access patterns

(Section 4.1.1)—which is why these two units alone account for 23 fields each in Table 4. The remaining fields select a shared unit's function or a post-merge multiplexer's input. Most of these values are compile-time constants baked in by the compiler as immediates, so the store is the entire cost; the handful of fields that depend on runtime state (e.g., a tensor's allocated base address, or a `:Dyn` query's clamped chunk size) additionally cost a few address-arithmetic cycles ahead of the store.

At the accelerator's 100 MHz operating clock, a single store-word issues in roughly 1–2 cycles (10–20 ns). The cheapest switch—updating only a shared configuration's multiplexer select, one field—costs about 20 ns; the most expensive supported operator, `Conv` (95 fields, Table 5), costs at most $95 \times 2 \approx 190$ cycles, or 1.9 µs, even if every field had to be rewritten from scratch.

Table 5. Per-operator configuration cost after merging.

	HW Units							Configuration	
Operator	**VRead**	**VWrite**	**F_AccumMax**	**F_Accum**	**Const**	**Buffer**	**Mux**	**Fields**	**Bits**
Add	2	1	0	0	0	-	-	69	1097
MatMul	2	1	0	1	0	-	-	71	1123
Conv	3	1	0	1	1	-	-	95	1524
AveragePool	1	1	0	1	0	-	-	48	754
MaxPool	1	1	1	0	0	-	-	48	754
BatchNorm	1	1	0	0	0	-	-	46	728
Relu	1	1	0	0	0	-	-	46	728
Post-Merge Mux and Buffers	-	-	-	-	-	4	6	10	58
Merged Accelerator	3	1	1	1	1	4	6	107	1608

Crucially, this can happen while the previous configuration is still running: the Config Reg accepts new field values immediately, while the delay chains ensure they only reach the Config Shadow Reg and thus only take effect once the corresponding phase is between runs. The right-hand side of Figure 8 illustrates why this is safe with a CPU-pipeline-style schedule: since Read, Compute, and Write are separate, delay-matched stages, up to three runs are in flight at once (e.g., Run 3 reads while Run 2 computes and Run 1 writes), so each phase's fields can be swapped in the moment that phase drains, independently of the other two, hiding the reconfiguration cost above behind the execution of the configuration it replaces. Even without that overlap—for instance, the very first switch, before any run is in flight to hide behind—the exposed cost stands on its own: sub-2 µs in the worst case, negligible next to any realistic operator execution time.

4.2. ONNX Parser

The ONNX Parser, implemented in Python using the `onnx` [3] and `onnxruntime` [53] libraries, loads the model, runs shape inference to resolve all tensor dimensions, and validates the constituent operators of the neural network Data Flow Graph (DFG) against the hardware capabilities. It emits a binary `CompiledModel struct` of serialised operations and parameters, together with C header artifacts. During parsing, each ONNX operator is matched against the operators listed in the Versat Spec file to determine whether it can be executed on the accelerator or must be handled by the RISC-V CPU. The ONNX Parser also outputs the model's parameters (inputs and weights) and a binary golden reference to validate the generated HW/SW system. Figure 9 illustrates the parser's pipeline, along with the generated artifacts by each stage.

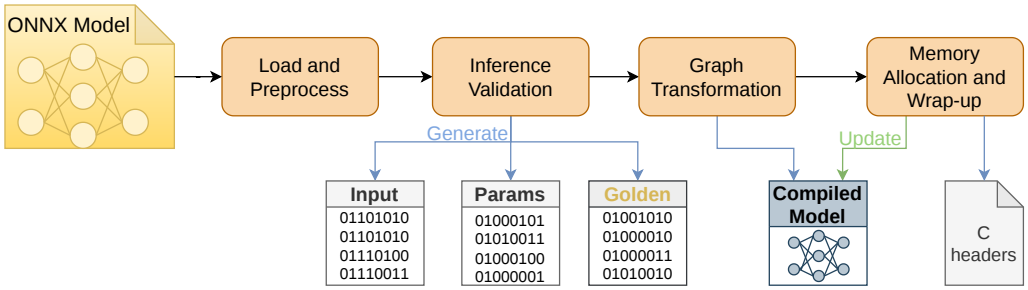


Figure 9. Multi-stage ONNX parser structure.

4.2.1. Load and Preprocess

The pipeline begins by loading the ONNX `protobuf` files and running the standard ONNX shape-inference pass to resolve all tensor dimensions. Initialisers are separated from live graph inputs (as required by ONNX IR 4 and above), and the model structure is validated against the ONNX specification. No compilation artifacts are produced at this stage.

4.2.2. Inference Validation

Before a model is trusted as a reference for hardware validation, the parser first checks that the model itself is sound, if the model is distributed with its own bundled test vectors: a sample input together with its expected output, as computed independently of Versat-AI. The preprocessed model is executed through ONNX Runtime on that bundled input to produce a floating-point golden reference, which is compared against the bundled expected output with a relative tolerance of 10^{-5}; any mismatch aborts the pipeline, since a model on ONNX Runtime cannot reproduce correctly on its own test data and cannot serve as a trustworthy reference for validating Versat-AI's hardware output later on. Only once this check passes is the golden reference computed here carried forward, alongside the model, for that later hardware validation.

4.2.3. Graph Transformation

Every ONNX operator supported by Versat-AI must provide a software-only implementation, executable on the RISC-V CPU, and may additionally provide a CGRA implementation via the Versat Spec file. Each node in the ONNX DFG is matched against this operator registry: if a CGRA implementation is available, the node is scheduled onto the accelerator; otherwise, if only a software implementation exists, the node falls back to CPU execution; operators with neither cause an immediate compilation error, giving the user a clear indication of which operators are entirely unsupported. For nodes that are scheduled, tensor shapes, operator attributes (strides, padding, dilation, etc.), and data-source classifications (model weight, live activation, or another node's output) are extracted and assembled into an intermediate representation that drives the subsequent stages.

At this stage we also perform graph optimisations that change the data format into an implementation that is easier to accelerate. The end result is the same, thus ensuring that these optimisations do not affect the correctness of results.

One example is related to the MatMul operation. In a normal implementation of MatMul, iterating the columns of the right-hand side matrix is costly since they are stored in row-major order. MatMul is an operation that regularly uses constants as the right side matrix. This gives us an opportunity to implement a simple optimisation: instead of iterating the columns, we can iterate the rows of the transposed matrix.

By storing the constant already transposed and by setting an attribute on the operator that instructs that the right side matrix is already transposed, we can use a faster MatMul implementation while producing the same results.

4.2.4. Memory Allocation and Wrap-Up

A liveness pass determines, for each intermediate tensor, the last graph node that consumes it. A greedy interval-fitting algorithm then packs live tensors into a single scratch-memory region, reusing space as soon as a tensor's lifetime ends. All allocations are 4-byte aligned to match the float32 element size. Permanent outputs are placed in a separate, sequentially allocated output region. The resulting offsets are recorded in the compiled model and used by the firmware runtime to address the on-chip memory.

The compiled model is serialised into a binary `CompiledModel` struct consumed by the firmware runtime, together with a model-independent C header that defines operator-type enumerations, per-operator parameter structs with variable-length trailing arrays, and accessor macros. The weight tensors and the floating-point golden reference generated in the validation stage are also written to the memory image, providing the firmware loader with all data required to initialize the accelerator. Once inference completes, the firmware compares Versat-AI's computed output element-wise against this embedded golden reference, using the same 10^{-5} relative tolerance as the earlier ONNX Runtime check, and reports a pass/fail result. This validation runs identically whether the firmware executes in host-CPU emulation or on the physical FPGA target (Section 4.3.2), since both run the same code against the same golden reference.

4.3. SoC Integration Framework

The `VersatAI.py` module describes the complete SoC as a Py2HWSW `iob_core` subclass, combining module definitions—ports, internal wires, and instantiated child modules—with instance properties such as parameter overrides and port-to-wire mappings. This allows the entire system to be composed from reusable building blocks without hand-written integration glue. The generated Versat-AI system is illustrated in Figure 10.

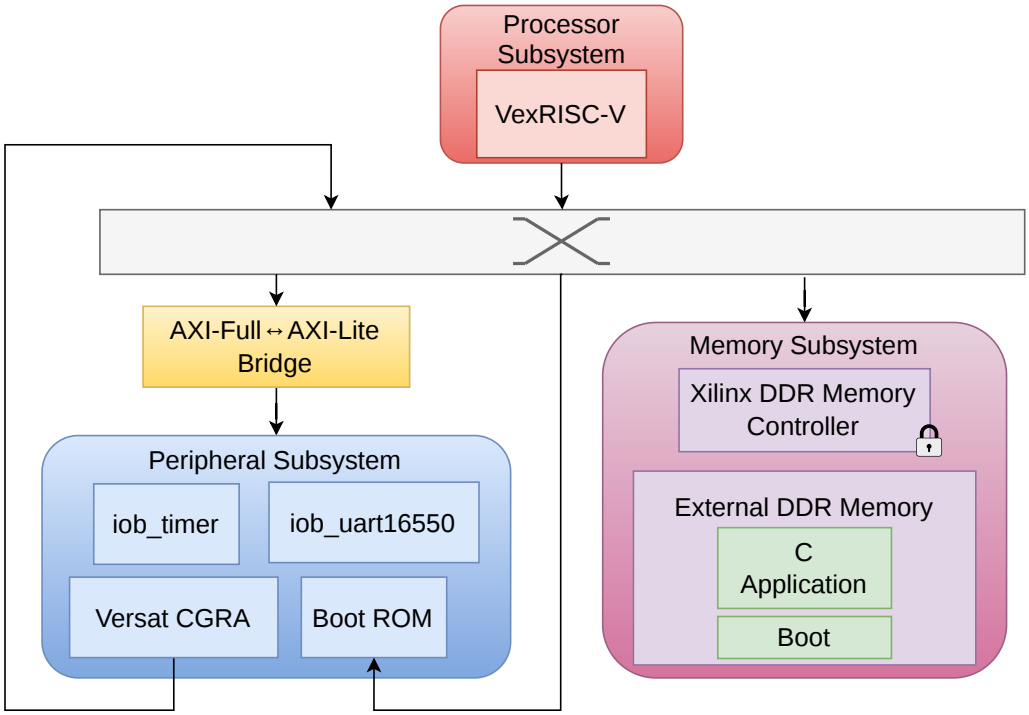


Figure 10. Versat-AI system.

4.3.1. CSR and Software Artifact Co-Generation

For every peripheral instantiated in the system description, Py2HWSW generates a matched hardware/software pair from the same Python source. On the hardware side

it produces a Verilog CSR module exposing each control register over the peripherals' bus. On the software side it produces a C header with macro definitions and a bare-metal driver with read/write accessor functions. For Versat-AI, the CSR block exposes the CGRA configuration register—operator-select, AGU parameters, and run-control signals—whose addresses and field layouts are kept in exact alignment with the firmware that the ONNX parser generates, since both are derived from the same Python description. The firmware runtime assembles these CSR writes at inference time from the `CompiledModel` struct emitted by the parser, configuring the CGRA for each operator in the execution schedule without any manually written glue code.

4.3.2. Unified Build and Emulation

Py2HWSW's hierarchical Makefile exposes simulation, FPGA synthesis, and host emulation from a single configuration .py file. An emulation mode compiles the SoC firmware for the host CPU and replaces hardware CSR accesses with memory-mapped reads and writes into a Verilator model, providing a co-simulation environment in which the same application runs on both host emulation and the FPGA target. This target-agnostic flow eliminates separate board support packages and ensures that simulation-passing firmware is byte-for-byte identical to what executes on the Kintex UltraScale KU040.

5. Experimental Results

Versat-AI was evaluated end-to-end on an Avnet AES-KU040-DB-G development board (Avnet, Phoenix, AZ, USA), built around a Xilinx Kintex UltraScale XCKU040-1FBVA676 FPGA (AMD, formerly Xilinx, Santa Clara, CA, USA) with 1 GB of DDR4 memory on a 32-bit bus at 1600 MT/s (6.4 GB/s theoretical peak), running at 100 MHz, using four MLPerf Tiny [6] inference tasks as a representative cross-section of edge workloads spanning image classification, anomaly detection, keyword spotting, and person detection. The evaluation is organised to answer four questions in turn. Section 5.1 asks whether the accelerator's silicon footprint is really independent of the evaluated models, as the compiled unit-merge strategy claims, and at what cost in on-chip memory. Section 5.2 checks that the compiler's own generated software is a fair, unhandicapped CPU baseline, and delimits what that baseline does and does not establish, before it is used to measure accelerator speedup. Section 5.3 then reports that speedup and situates it against related CGRA and dataflow accelerators. Section 5.4 presents a power and energy analysis, translating the measured speedup into energy per inference under the system's post-implementation estimated power budget. All of these results describe the current, single-operator-instance phase of the project, in which each operator runs on a single physical datapath instance rather than several in parallel; the limitations of that phase, and the outlook beyond it, are discussed separately in Section 6.

5.1. FPGA Resource Utilisation

The resources utilised by the generated system are presented in Table 6. In this version, a single instance of each floating-point operator is used. However, keep in mind that the plan is to have multiple fixed-point instances to scale the performance while keeping the overall footprint predictable and modular.

Compared to some related works, the Versat-AI accelerator footprint is independent of model size. FINN allocates a dedicated hardware pipeline per network layer, so LUT and BRAM usage grows with the number of layers [25]. Gemmini employs a reusable systolic array that amortises computation across layers, but it primarily targets ASIC flows and requires a large FPGA to host the full RISC-V system alongside the array [21]. Versat-AI's unit-merge strategy instead collapses all operator variants into a single CGRA, holding the

accelerator to 8763 LUTs and 4 DSPs for the full operator set regardless of model size, a key advantage for resource-constrained edge deployments.

Table 6. Resource utilisation for the XCKU040 FPGA.

	Component	LUTs	FFs	DSPs	BRAMs
Versat-AI	Versat Accelerator	8763	9833	4	202
	VexRISC-V	3062	2571	4	5
	DDR4 Controller	10,728	13,740	3	26
	Other periph. & bus	4984	4361	0	1
	Total	27,537	30,505	11	234
Base SW	VexRISC-V + FPU	5074	4503	7	5
	DDR4 Controller	10,714	13,740	3	26
	Other periph. & bus	2855	4343	0	1
	Total	18,643	22,586	10	32

On-Chip Memory Cost

The merge strategy is markedly less economical in block Random Access Memory (RAM) than in logic. The accelerator consumes 202 Block Random-Access Memories (BRAMs) against the Base SW system's 32, so while it adds only 8763 LUTs and 4 DSPs, it multiplies the system's on-chip memory requirement by more than six. On the XCKU040's 600 BRAM tiles this is 34% of the device—a larger fraction than the accelerator occupies of any other resource type—and it makes BRAM, not logic, the binding constraint when porting the design to smaller parts.

This cost is structural and the hierarchical utilisation report locates it precisely. Of the 202 tiles, 189 (94%) are the external ping-pong buffers of the three `VRead` and one `VWrite` instances of the merged datapath (Table 5): the three `VRead` buffers occupy 58 tiles each and the `VWrite` buffer 15, the asymmetry reflecting that the merged read ports must be sized for the largest feature-map or weight tile any operator streams, whereas the write port drains a single output tile. The remaining 13 tiles are the CGRA's four internal pipeline buffers (1 tile each) and 9 tiles of control and configuration memory; the merged datapath's arithmetic units themselves consume no BRAM at all. As described in Section 4.1.2, these buffers exist precisely to decouple the two access patterns the design must reconcile: DDR is efficient only under sequential bursts, whereas the datapath requires one value per cycle under linear, strided, broadcast, or 2-D windowed addressing. The buffer is what converts one into the other, and it is also what allows a DMA transfer for the next tile to overlap with computation on the current one, which is what keeps the datapath from being data-starved (Section 3.1).

Buffer depth is therefore a direct trade against performance, and the `:Dyn` folding mechanism of Section 4.1.1 makes that trade continuous rather than a correctness boundary: when a tensor exceeds the buffer, the compiler folds the loop and issues the operator over several passes. Halving the buffers would halve the BRAM count at the cost of roughly doubling the number of passes for the operators that already fold, each pass re-fetching its operands from DDR. Since the DDR controller already accounts for 41% of system power (Section 5.4) and inference here is bandwidth rather than compute-bound, BRAM bought back this way is paid for in both latency and energy. The favourable property is that, like the logic footprint, the BRAM cost is fixed with respect to model size: it is set by the operator vocabulary and the chosen buffer depth, so it is a one-time cost rather than one that grows with network depth as in FINN. The measured breakdown also bounds what any such trade can recover: since 94% of the cost is in four buffers whose depth is a generation-time parameter, BRAM is tunable almost in its entirety, but only along this

latency axis. A quantitative sweep of buffer depth against inference latency and energy, which would identify the knee of this curve for each benchmark, has not been performed and is left as future work; the measurements above establish where the cost sits and that it is parametric, and the argument above its direction but not its slope.

5.2. Software Generation Quality

Before using our own compiler's generated software as the baseline for accelerator speedup, we first check that it is a competent, representative CPU implementation rather than an artificially weak one—since an accelerator's apparent speedup is only meaningful if measured against a fair software baseline. Table 7 compares the execution time of C code generated by our ONNX parser (Base SW) against ONNX2C [54], an independent, third-party transpiler that converts ONNX models directly to C, on the same VexRISC-V CPU for the four evaluated benchmarks; the Ratio column is the ONNX2C time divided by the Base SW time, so values above 1 indicate Base SW is faster.

Table 7. Software generation quality: execution time (seconds), ONNX2C vs. Base SW.

Benchmark	ONNX2C	Base SW	Ratio
ResNet/CIFAR-10	5.08	6.59	0.77
Dense/ToyCar	0.18	0.09	2.00
DS-CNN/Speech	1.45	2.05	0.71
MobileNet/VWW	10.33	5.85	1.77

Base SW is faster than the independent ONNX2C baseline on two of the four benchmarks and slower on the other two, with no consistent bias in either direction. This indicates Base SW is a reasonably competent, unhandicapped software implementation rather than a deliberately weak strawman, which is the property that matters for using it as the baseline in the speedup comparison that follows: unlike a comparison against ONNX2C, where an accelerator's apparent speedup could partly reflect ONNX2C's own code generation choices rather than the accelerator's actual contribution, comparing against Base SW isolates the effect of the accelerator alone, since both the software-only and accelerated paths share the same compiler-generated software backbone.

Two qualifications on this baseline must be stated explicitly, since they bound how the speedups of Section 5.3 should be read. First, the margins are not small in the two cases where Base SW loses: it is 29% slower than ONNX2C on ResNet and 41% slower on DS-CNN. A baseline that is one-third slower than an independent transpiler on half the benchmarks is adequate evidence of the absence of a strawman, but it is not evidence of parity. Rather than leave the consequence of this to the reader, the accelerator is measured against both baselines separately in Section 5.3: Table 8 reports the speedup over Base SW, and Table 9 reports the speedup over ONNX2C, so the result obtained against the faster software implementation on each benchmark can be read directly. Second, and more fundamentally, neither Base SW nor ONNX2C is an optimised embedded inference runtime. Both are portable C generators compiled for a VexRISC-V softcore; neither exploits the hand-tuned, architecture-specific kernels that libraries such as PULP-NN [19] or CMSIS-NN [55] provide, nor the quantised arithmetic those libraries assume. The 2.3×–9× figures that follow are therefore speedups over the software path of this same flow—the internally controlled comparison that isolates the accelerator's contribution—and are not claims of a margin over the best available software implementation for embedded inference. Section 6.1 discusses why a direct head-to-head measurement against such implementations is not currently available.

Table 8. Execution time (seconds) and speedup over the Base SW baseline generated by this flow.

Benchmark	Base SW	Versat-AI	Speedup
ResNet/CIFAR-10	6.59	2.91	2.3×
Dense/ToyCar	0.09	0.01	9×
DS-CNN/Speech	2.05	0.61	3.4×
MobileNet/VWW	5.85	1.07	5.5×

Table 9. Execution time (seconds) and speedup over the independent ONNX2C baseline.

Benchmark	ONNX2C	Versat-AI	Speedup
ResNet/CIFAR-10	5.08	2.91	1.7×
Dense/ToyCar	0.18	0.01	18×
DS-CNN/Speech	1.45	0.61	2.4×
MobileNet/VWW	10.33	1.07	9.7×

5.3. Performance

A subset of the `MLPerf Inference:Tiny` benchmark [6] was used for evaluation: image classification with ResNet [56] on CIFAR-10 [57], anomaly detection with a dense autoencoder on the ToyADMOS dataset [58], keyword spotting with DS-CNN [59] on Speech Commands [60], and person detection with MobileNet [61] on Visual Wake Words [62]. The accelerator is measured against both software baselines separately, in Table 8 against the Base SW produced by this flow and in Table 9 against the independent ONNX2C transpiler, so that the speedup obtained is reported against each reference rather than only against the more favourable one. All three configurations run on the same VexRISC-V core, but the two CPU-only configurations require a hardware FPU to perform the inference in software, which roughly doubles the CPU's resource footprint compared to the FPU-less VexRISC-V used alongside the accelerator in the Versat-AI system, where the accelerator itself performs all floating-point computation. In each table the speedup is the baseline time divided by the Versat-AI time.

The accelerator is faster than both software baselines on all four benchmarks, so the direction of the result does not depend on which reference is chosen—the magnitude does. Measured against Base SW the speedups span 2.3×–9×; measured against ONNX2C they span 1.7×–18×, falling on the two benchmarks where ONNX2C is the faster software implementation (ResNet, DS-CNN) and rising on the two where it is slower (ToyCar, MobileNet). The conservative reading of these results is therefore the per-benchmark minimum of the two tables—1.7× on ResNet, 2.4× on Speech, 5.5× on MobileNet and 9× on ToyCar—which is the speedup retained against the faster of the two available software implementations in every case. The ToyCar ratios should be read as approximate in both tables, since a 10 ms accelerator time leaves only one significant figure in the denominator.

Since Versat-AI maps operators onto a fixed set of merged hardware units, accelerator area is independent of network size: the same fixed-footprint accelerator instance, with no resynthesis or area change, was used to run all four evaluated models.

Versat-AI's own speedup can also be set alongside related CGRA and dataflow accelerators, though not as a head-to-head ranking: what is compared here is the present single-operator-instance Versat-AI, which instantiates a single physical datapath per operator, against these works' own reported figures, some of which use parallel operator replication. Within that scope, CONVOLVE's CGRA path reaches an average 3.6× speedup over its RISC-V software baseline [27]. Since that figure is measured against the authors'

own software baseline, the like-for-like Versat-AI numbers are those of Table 8. Despite its single-operator-instance accelerator, Versat-AI matches or exceeds that figure on two of four benchmarks (9× on ToyCar, 5.5× on MobileNet), against 2.3× on ResNet and 3.4× on Speech. The lower speedup on ResNet follows from its higher arithmetic intensity: standard 3 × 3 convolutions reuse each loaded weight many times, so its CPU baseline is closer to compute-bound and the bandwidth-disciplined accelerator has less memory traffic to exploit. The autoencoder and MobileNet's depthwise-separable convolutions reuse weights far less, placing them in the memory-bandwidth-bound regime this design targets.

SNAFU and RipTide report 9.9× and 6.2× improvements, respectively, but on general dataflow programs rather than MLPerf Tiny tasks and without a direct ONNX front end or full SoC generation [15,16]. FINN and Gemmini operate on quantised integer models and target different benchmark suites, making direct latency comparison impractical: FINN's hardware footprint scales with model depth and width, and while Gemmini's Chipyard integration and CGRA4ML [23] each produce a complete SoC around their accelerator, both build it around a fixed, designer-sized array template rather than one a compiler derives automatically from an arbitrary ONNX model—and CGRA4ML's front end further accepts only QKeras-trained models rather than a standard interchange format [21,23,25].

These comparisons should be read with the qualification stated in Section 5.2: the speedups are measured against the software path of this same flow, not against an optimised embedded inference runtime, and Section 6.1 sets out why a direct head-to-head measurement against quantised edge-AI frameworks is not currently available. Closing the remaining throughput gap without giving up the flow's generality is the target of the parallel operator instancing outlined in Section 6.3.

5.4. Power and Energy Analysis

Post-implementation power analysis using Xilinx Vivado estimates the complete SoC at 1.96 W at 100 MHz, as broken down in Table 10.

Table 10. Post-implementation power breakdown (Vivado estimate, 100 MHz).

Component	Base SW (W)	Versat-AI (W)	Increase
Versat Accelerator	–	0.568	–
DDR4 Controller	0.804	0.809	1.0×
VexRISC-V CPU (+FPU for Base SW)	0.058	0.037	0.6×
Peripherals & bus	0.033	0.046	1.4×
Total Dynamic	0.895	1.460	1.6×
Total Power	1.383	1.959	1.4×

The DDR4 controller accounts for 41% of total system power, reflecting the memory-bandwidth-bound nature of floating-point Deep Neural Network (DNN) inference. The Versat accelerator itself draws only 0.65 W—33% of estimated system power—despite performing all operator-level computation.

Because system power is approximately constant within each configuration, energy per inference scales directly with latency, computed as $E = P_{\mathrm{SoC}} \times t$, using the post-implementation estimated total power from Table 10 for each configuration (1.383 W for Base SW, 1.959 W for Versat-AI). Table 11 reports the resulting energy per inference for both configurations.

The energy reductions are smaller than the raw speedup ratios because the software-only SoC also draws less power than the full Versat-AI SoC (Table 10): the energy reduction equals the speedup scaled by the inverse of the 1.4× power increase shown there (≈0.71).

Even so, energy reduction still tracks latency reduction directly within each configuration, and because the merge strategy ties the accelerator's footprint to the operator vocabulary rather than model size, these savings hold across all evaluated models regardless of size. An ASIC implementation at a sub-28 nm node would reduce the accelerator contribution substantially: prior work on dataflow accelerators of comparable complexity reports active power under 10 mW at equivalent throughput [15,16], suggesting that the dominant system-level power term would shift from compute to memory I/O even more strongly. Future fixed-point quantisation will reduce the DDR4 traffic directly, targeting an overall system energy reduction beyond the speedup gains alone.

Table 11. Energy per inference.

Benchmark	Base SW (J)	Versat-AI (J)	Reduction
ResNet/CIFAR-10	9.13	5.70	1.6×
Dense/ToyCar	0.125	0.020	6.4×
DS-CNN/Speech	2.84	1.20	2.4×
MobileNet/VWW	8.10	2.10	3.9×

6. Discussion and Future Directions

Section 5.3 establishes what this phase of the project set out to demonstrate: an arbitrary ONNX model yields a running FPGA system whose accelerator footprint is set by the operator vocabulary rather than by the model, at a 2.3× to 9× latency advantage over the software path of the same flow. Two design choices bound that result—single precision arithmetic and a single datapath instance per operator—and both also determine which external comparisons are currently meaningful. Each is examined below, together with the path to lifting it.

6.1. Scope of the Experimental Comparison

Versat-AI is evaluated against the software path of its own flow, a controlled experiment by construction: both configurations share the compiler front end, the VexRISC-V core, the memory subsystem, and the board, and differ only in the presence of the accelerator. The measured ratio is therefore attributable to the accelerator itself, which is what a first evaluation of an automatically derived datapath needs to isolate.

A direct comparison against other edge-AI frameworks measures a different property—throughput ranking rather than the contribution of the derived datapath—and three conditions must change before it can be made on comparable terms.

The first is arithmetic format. FINN [25], Gemmini's ONNX flow [21], CGRA4ML [23], and the PULP family [18,19] operate on quantised integer models—in FINN's case down to binary and ternary weights—whereas Versat-AI executes FP32 (Section 6.2). Across that boundary a latency ratio measures operand width at least as much as architecture: an INT8 design moves one quarter of the bytes per operand over a memory-bound datapath and packs several MACs into a single DSP slice.

The second is the platform. CGRA4ML reports on a taped-out ASIC with an Arm host, Gemmini principally targets ASIC flows on a Rocket host, and GAP8/GAP9 are 55 nm and 22 nm ASICs rather than FPGA overlays. Normalising across technology node, host processor, clock, and memory subsystem introduces more uncertainty than the comparison resolves. Where the methodology does align, the comparison is made: CONVOLVE [27] reports speedup against its own RISC-V software baseline, and Section 5.3 compares against that figure directly, Versat-AI matching or exceeding CONVOLVE's 3.6× average on two of four benchmarks with a single datapath instance per operator.

The third is workload admissibility. FINN and hls4ml [26] require the models retrained or quantisation-aware converted; CGRA4ML accepts only QKeras-trained models; Vitis AI [28] targets DPU overlays sized for mid-to-high-density devices and is not open below the library level. The MLPerf Tiny models used here do not pass through those flows unmodified, so a measurement obtained on the same XCKU040 board would compare different workloads.

Quantisation (Section 6.2) removes the first and largest of the three: once the datapath executes INT8, the same models pass unmodified through the quantised flows and a same-board, same-model, same-format comparison becomes available, and establishing it is the natural continuation of this work. Such a comparison ranks throughput, a property orthogonal to the one evaluated here and offered by none of the frameworks above: accelerator, drivers, and application software derived automatically from an unmodified ONNX model, at a footprint fixed by the operator vocabulary rather than by the network.

6.2. Numerical Precision and the Path to Quantised Inference

The datapath is single-precision floating-point throughout (Table 2). FP32 lets the generated system be validated element-wise against ONNX Runtime's own execution of the unmodified model at a 10^{-5} relative tolerance (Section 4.2), so any discrepancy is attributable to the compiler or the datapath rather than to quantisation error. With the automation demonstrated end to end, that validation anchor has served its purpose, and integer arithmetic—the norm at the edge, where Post-training Quantisation (PTQ) and Quantisation Aware Training (QAT) are applied as a matter of course [1]—is the next priority. The gain accrues on three fronts.

In area, an INT8 MAC maps to a fraction of a single DSP slice, and UltraScale DSP slices pack two 8-bit multiplications each, so several integer MACs fit in the area that one FP32 unit occupies in Table 6. In on-chip memory, narrower operands reduce the streaming-buffer occupancy analysed under *On-Chip Memory Cost* in Section 5.1 proportionally, relieving the BRAM cost identified there. In energy, the effect is decisive: the DDR4 controller accounts for 41% of system power (Table 10), and off-chip memory access dominates the energy of DNN inference by a wide margin over arithmetic [1]. Moving to INT8 cuts operand traffic by four times at its source, improving system energy by more than the arithmetic savings alone, and relaxing the bandwidth ceiling that Section 6.3 identifies as the limit on parallel scaling.

The existing structure localises the required changes. On the parser side (Section 4.2), ONNX represents quantised models through the `QuantizeLinear`, `DequantizeLinear`, and `QLinear*` operator families, carrying per-tensor or per-channel scale and zero-point initialisers. The parser already extracts operator attributes and classifies each tensor's data source, so scale and zero-point become additional attributes on the existing intermediate representation; the memory allocator's 4-byte alignment assumption and element-size arithmetic generalise to 1- and 2-byte elements, and the graph-transformation stage folds quantise/dequantise pairs into the neighbouring operator's parameters instead of emitting them as nodes. The golden reference becomes ONNX Runtime's own quantised execution, with an exact or integer-tolerance match replacing the 10^{-5} relative tolerance.

On the hardware side, the change is confined to the functional-unit library. The merge algorithm of Section 4.1.3 operates on unit types, ports, and latencies, and is indifferent to the width or interpretation of the data a unit carries; the AGU, timing-correction, and configuration machinery are likewise format-agnostic. Integer units therefore extend the operator vocabulary the merge sees without altering how the merge is computed. Two additions are new: a requantisation stage—a multiply, shift, and saturating clamp applied to accumulator output—as a functional unit in its own right, and accumulators widened to

32 bits over 8-bit operands to avoid overflow. The `versat_latency` attribute mechanism means each new unit declares its own pipeline depth and is scheduled automatically. Quantisation support thus reaches the unit library and the parser's type handling and leaves the merge algorithm, the timing correction, and the SoC integration untouched.

6.3. Outlook: Parallel Operator Instancing

The reported results use a single physical datapath instance per operator, processing one element per cycle over a 32-bit memory port—the most conservative configuration the flow can generate. The next step keeps each operator's datapath unchanged and instantiates it N times in parallel, at the same granularity as YOLO-Versat's compute array [8], with the compiler sizing N from each model's widest layer and the available bandwidth. This extends the automation from sizing the operator vocabulary to sizing its parallelism.

Two measured anchors bound the room available. Versat-AI's target board (Avnet AES-KU040-DB-G) carries 1 GB of DDR4 memory on a 32-bit bus at 1600 MT/s, a 6.4 GB/s theoretical peak, and YOLO-Versat, on the same FPGA family, sustains 228 GOPS from an 832-MAC array using 44% of the device's DSP budget [8], placing device capacity on the order of 10^3 parallel MACs. This bound characterises the device alone: occupying it would require the fixed-point arithmetic of Section 6.2, both to fit the units and to narrow the operand traffic feeding them, together with an explicitly planned dataflow and a multi-banked buffer hierarchy whose tiling and refill schedules are co-designed with the array geometry—design effort that a hand-tuned single-network accelerator such as YOLO-Versat can amortise over one fixed topology but that a generated, model-agnostic datapath cannot assume. A more proportionate first step replicates the existing datapath while retaining the present single-tile buffer structure, widened rather than restructured, which places N in the range of 10^1 to low 10^2 MACs at a small fraction of that effort. For the low-arithmetic-intensity operators this design targets, bandwidth saturation rather than MAC count sets achievable utilisation, so that lower range may already reach the point at which additional instances stop yielding throughput; locating that point is a central objective of the work rather than an assumption behind it, and it determines whether the buffer redesign that the 10^3 regime requires is worth undertaking at all.

A first-order instantiation of that lower range follows directly from each operator's own memory-access pattern rather than from the device bound alone. Expressed in the accelerator's native 32-bit-word units rather than bytes, the 6.4 GB/s DDR4 peak at 100 MHz is 16 words per cycle—16 floating-point values read and written, combined, in the best case. `Relu` reads and writes exactly one value per unit per cycle, so 8 parallel units read 8 and write 8, exactly saturating that budget. `Add` reads two operands and writes one per unit; 6 units read 12 and write 6, 18 words against the 16-word budget, oversubscribing it in the worst case, though ONNX's broadcast semantics let one operand be reused across outputs rather than re-read every cycle, which can absorb some of the excess. `MatMul` also reads two operands per output nominally, but one of them—typically the weight matrix—is reused across many outputs rather than re-fetched, so its average consumption is closer to 1.5 reads per output; 6 units then read 9 and write 6, 15 words, just under budget. `Conv` does not fit this read/write accounting: a $1 \times 3 \times 3$ kernel requires 9 MACs per output, and because accumulating floating-point partial sums is comparatively expensive in this datapath, 4 such units—36 MACs in total, 4 outputs per cycle—are taken as a practical upper bound rather than one derived from the bandwidth budget directly. These four counts—8 `Relu`, 6 `Add`, 6 `MatMul`, and 4 `Conv` (36 MAC)—sit at the low end of the 10^1-to-low-10^2 range above and are offered as an order-of-magnitude sizing rather than a final one: they assume floating-point operands throughout and the present single-tile buffer

structure, both of which the fixed-point path of Section 6.2 and a wider buffer hierarchy would revise.

Sizing N this way also sizes the buffers behind it, because self-reconfiguration (Section 3.1) lets Versat-AI reconfigure while a previous run still executes, which makes reconfiguration latency, not buffer capacity, the quantity that sets the minimum useful run length: a run shorter than the time it takes to reconfigure for the next one wastes cycles waiting rather than computing, while a longer run only stretches the same amortisation without buying anything more. For the parallel datapath above, reconfiguring an operator means rewriting the CSR fields of all N of its units together with the run's runtime-computed values (e.g., base addresses) and any operator-specific setup logic, rather than just the single unit's fields that the 190-cycle worst case for the present single-instance datapath (Table 5) bounds; taking this at roughly 4000 cycles—an order-of-magnitude estimate, since the exact figure is operator- and N-dependent—means each run should process on the order of 4000 values to keep reconfiguration overhead small relative to execution. Rounding to 4096, each buffered port must hold that many floating-point values, 16 KB, doubled to 32 KB by the ping-pong scheme that overlaps configuration with execution. Because `VRead` and `VWrite` instances are merged across the datapath (Section 4.1.3), the total requirement is set by the worst-case operator rather than the sum of all of them: `Add` and `MatMul` each need two 32 KB input buffers and one 32 KB output buffer per unit, and at 6 units this is $6 \times 3 \times 32\,\text{KB} = 576\,\text{KB}$, or approximately 144 BRAM tiles—24% of the XCKU040's 600-tile budget, comfortably within the device though not negligible against it. Both this figure and the reconfiguration-cycle estimate it rests on move together: doubling one doubles the other, and either would fall if the fixed-point narrowing of Section 6.2 is applied first.

Taken together, this sizing points to a rough 8× speedup over the current single-instance datapath, driven mostly by `Conv`—whose nine-cycle-per-kernel cost under one-value-per-cycle memory access is the largest single contributor to current runtime—and by the proportionally faster `Add` and `Relu` stages once they are no longer serialised behind a single 32-bit port. Like the unit counts and buffer size above, this is a back-of-envelope estimate from the operators' own access patterns, not a synthesised or measured one, and assumes a model with layers large enough to keep the parallel units and the 4096-value runs above actually fed.

The four result tables respond to N differently, and none scales with it in the same way. Resources (Table 6) split into two regimes. The 8763 LUTs and 4 DSPs of the merged datapath are per-instance costs and grow approximately linearly with N, which is what the 10^3-MAC device capacity bounds. BRAM does not split as cleanly. The 202 BRAMs of the present single-instance design are 94% streaming buffers (see *On-Chip Memory Cost* in Section 5.1) sized by the largest operand tile a single merged `VRead/VWrite` instance must stream, a cost paid once and shared across every operator that uses it. If each of the N replicated units instead keeps its own ping-pong buffer, sized for reconfiguration amortisation as in the sizing above—576 KB, or 144 BRAM tiles, at $N = 6$ for the Add/MatMul case alone—that per-instance cost no longer stays flat with N but grows with it, tracking the LUTs and DSPs rather than diverging from them. Whether that per-unit buffering can instead be shared, with N narrower ports drawing from one buffered tile rather than N private copies, is an open buffer-hierarchy question this projection does not resolve, and it is what ultimately decides whether BRAM or LUT/DSP occupancy ends up the dominant constraint as N grows.

Performance (Tables 8 and 9) improves sub-linearly in N, under two independent ceilings. The first is compositional: whole-inference speedups are bounded by the fraction of each benchmark's runtime spent in accelerated operators, and the residue on the RISC-V core is untouched by N. The second is the bandwidth limit already stated: once the 6.4 GB/s

DDR4 peak saturates, further instances change utilisation rather than latency. Benchmarks with the largest share of runtime in `Conv` and `MatMul` gain most, so the spread between the four widens rather than shifting uniformly.

Power (Table 10) rises by less than N, and energy (Table 11) falls. Static power is a device property and is essentially unchanged. Dynamic logic power grows with the replicated arithmetic, but the DDR4 share—41% of the reported total—does not grow proportionally, because it is already at the ceiling that limits the scaling in the first place: the same operand traffic is delivered in less time rather than in greater volume. Energy per inference is the figure most likely to improve outright, since the dominant off-chip term is roughly conserved per inference while static power is paid over a shorter runtime.

These are qualitative directions rather than predicted values, and the unit counts, buffer size, and 8× figure derived above are order-of-magnitude estimates from the operators' own access patterns rather than synthesised or measured results; their realisation and measurement remain future work.

7. Conclusions

This paper presented Versat-AI, an open-source ONNX-to-SoC compiler that automatically derives a complete CGRA-based edge inference system—accelerator, firmware, and application software—from an arbitrary ONNX model. The central technical contribution is the application of the hardware merge strategy [4] to neural network operator mapping: structurally compatible ONNX operators are collapsed into a physical CGRA instance, so the accelerator's hardware directly reflects the operator vocabulary the input model actually uses, rather than a designer-fixed, model-independent template or a hardware pipeline allocated per layer. Integrated with the Py2HWSW [5] SoC generator, the system produces a complete synthesisable RISC-V SoC—RTL, memory-mapped interconnects, firmware drivers, and application software—from a single ONNX input, eliminating the manual effort of hardware/software co-design for edge inference.

The accelerator was validated on the Avnet AES-KU040-DB-G board's Xilinx Kintex UltraScale FPGA (XCKU040) at 100 MHz, accelerating seven ONNX operators in 8763 LUTs, 9833 FFs, 4 DSPs, and 202 BRAMs—a footprint that remained unchanged across all evaluated models regardless of size. The complete SoC totals 27,537 LUTs and 30,505 FFs at a Vivado post-implementation estimated 1.96 W, of which the accelerator accounts for only 0.65 W. The merge strategy ties both silicon footprint and power to the operator vocabulary rather than model size. Across the four MLPerf Tiny tasks [6] evaluated, execution time is 2.3× to 9× shorter than the software-only baseline generated by the same flow, and energy per inference is reduced by 1.6× to 6.4×—the smaller energy factor reflecting the 1.4× higher power of the accelerated SoC. These are the results of the current, single-operator-instance phase of the project, in which each operator runs on a single physical datapath instance over a 32-bit memory port, and they are evidence that the automatic derivation itself is sound. The compiler's next step is to extend that same automation from sizing the operator vocabulary to sizing per-operator parallel instancing and bandwidth, following the bandwidth-matched scaling principle already demonstrated, by hand, in YOLO-Versat—this accelerator's own design lineage. Among related flows, FINN [25] scales resource usage with network depth and Gemmini [21] targets ASIC flows on high-density FPGAs; CGRA4ML [23] compiles a complete taped-out SoC around a CGRA but from a fixed array template and a QKeras-specific front end rather than an unmodified ONNX model; CONVOLVE [27] reports a 3.6× average speedup on its CGRA path, a figure Versat-AI's single-operator-instance accelerator matches or exceeds on two of four benchmarks. None of these, however, derives a complete SoC—accelerator, firmware, and application software—automatically from an unmodified ONNX model with a footprint tied to the model's own operator vocabulary.

Three aspects delimit the scope of these results and are examined in Section 6: the datapath is single-precision floating-point, chosen so the generated system could be validated element-wise against ONNX Runtime's execution of the unmodified model; each operator runs on a single datapath instance, the most conservative configuration the flow can produce; and the accelerator's 202 BRAMs are a heavier cost than its 8763 LUTs, set by the depth of the streaming buffers that keep the datapath fed.

Each defines a line of future work. Fixed-point quantisation will reduce memory bandwidth and improve energy efficiency [1]—cutting the DDR4 traffic that currently dominates system power—and, by removing the numeric-format difference discussed in Section 6.1, will also enable a direct comparison against existing quantised edge-AI frameworks on the same board and the same models. The accelerator's 32-bit memory port will be widened toward the bandwidth-matched interfaces YOLO-Versat already validated, and N parallel copies of each operator's datapath instantiated, with the compiler automatically sizing their number to each model's widest layer, scaling throughput while preserving the tie between footprint and operator vocabulary rather than model size; establishing where bandwidth saturation caps that scaling is part of the same work. A first-order sizing from the operators' own access patterns (Section 6.3) places that scaling at 8 `Relu`, 6 `Add`, 6 `MatMul`, and 4 `Conv` units before the DDR4 bandwidth budget saturates, for a projected 8× speedup over the current single-instance datapath—an order-of-magnitude estimate to be confirmed by implementation, not a synthesised or measured result. A buffer-depth sweep will quantify the BRAM-versus-latency trade-off, and the design will be validated through an ASIC flow targeting a sub-watt embedded implementation. Added compute parallelism should particularly benefit compute-bound workloads such as ResNet's standard convolutions, where speedup was lowest in the present evaluation.

Finally, the background section traced the two-decade evolution of reconfigurable accelerators—from SideWorks and Versat through DeepVersat and the YOLO accelerator—that shaped Versat-AI's design, and documented the open-source SoC platform development, including integration attempts with the OpenRISC ecosystem [37] and the Rocket Chip generator [41], and the design decisions that led to IOb-SoC and Py2HWSW.

Author Contributions: Conceptualisation, R.T. and J.T.d.S.; methodology, R.T., J.B.R., J.A. and J.L.; software, R.T., J.B.R. and J.A.; validation, R.T., J.B.R. and J.A.; formal analysis, R.T., J.L. and J.T.d.S.; investigation, R.T., J.B.R. and J.T.d.S.; resources, J.T.d.S.; writing—original draft preparation, J.B.R. and J.A.; writing—review and editing, R.T., J.B.R., J.L. and J.T.d.S.; visualisation, R.T. and J.B.R.; supervision, J.T.d.S.; project administration, J.T.d.S.; funding acquisition, J.T.d.S. All authors have read and agreed to the published version of the manuscript.

Funding: This project was mainly funded by the NLnet NGI0 Core Fund, with support from the European Commission's Next Generation Internet programme, grant agreement No. 101092990. The project was partially funded by the A-IQ-Ready project/Chips JU, grant agreement No. 101096658. J.B.R. and J.A. acknowledge Fundação para a Ciência e a Tecnologia (FCT) support through grant BI|2025/772–Proj. 2023.16747.ICDT (SMARTChip, DOI: https://doi.org/10.54499/2023.16747.ICDT, Ref[a] LISBOA2030-FEDER-00692100), and projects UID/6486/2025, UID/PRR/6486/2025, UID/PRR2/06486/2025, UID/50021/2025, UID/PRR/50021/2025.

Data Availability Statement: The Versat-AI source code is openly available at https://github.com/IObundle/versat-ai (accessed on 24 August 2026).

Conflicts of Interest: Author Rúben Teixeira, Jaime Aguiar and José T. de Sousa were employed by the company IObundle, Lda. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Sze, V.; Chen, Y.H.; Yang, T.J.; Emer, J.S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* **2017**, *105*, 2295–2329. [CrossRef]
2. Liu, L.; Zhu, J.; Li, Z.; Lu, Y.; Deng, Y.; Han, J.; Yin, S.; Wei, S. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. *ACM Comput. Surv.* **2019**, *52*, 118. [CrossRef]
3. ONNX Community. ONNX: Open Neural Network Exchange. GitHub Repository. 2019. Available online: https://github.com/onnx/onnx (accessed on 24 August 2026).
4. De Sousa, J.T.; Martins, V.M.G.; Lourenco, N.C.C.; Santos, A.M.D.; Ribeiro, N.G.D.R. Reconfigurable Coprocessor Architecture Template for Nested Loops and Programming Tool. US Patent 8,276,120, 25 September 2012.
5. IObundle. Py2HWSW. 2026. Available online: https://github.com/IObundle/py2hwsw (accessed on 24 August 2026).
6. Banbury, C.; Reddi, V.J.; Torelli, P.; Jeffries, N.; Kiraly, C.; Holleman, J.; Montino, P.; Kanter, D.; Warden, P.; Pau, D.; et al. MLPerf Tiny Benchmark. In *Proceedings of the Thirty-Fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*; Curran Associates, Inc.: Red Hook, NY, USA, 2021.
7. Moreano, N.; Borin, E.; de Souza, C.; Araujo, G. Efficient datapath merging for partially reconfigurable architectures. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2005**, *24*, 969–980. [CrossRef]
8. Pestana, D.; Miranda, P.R.; Lopes, J.D.; Duarte, R.P.; Véstias, M.P.; Neto, H.C.; de Sousa, J.T. A Full Featured Configurable Accelerator for Object Detection With YOLO. *IEEE Access* **2021**, *9*, 75864–75877. [CrossRef]
9. Lopes, J.D.; de Sousa, J.T. Versat, a Minimal Coarse-Grain Reconfigurable Array. In *Proceedings of the High Performance Computing for Computational Science—VECPAR 2016*; Dutra, I., Camacho, R., Barbosa, J., Marques, O., Eds.; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2017; Volume 10150, pp. 174–187. [CrossRef]
10. Lopes, J.D.; Véstias, M.P.; Duarte, R.P.; Neto, H.C.; de Sousa, J.T. Coarse-Grained Reconfigurable Computing with the Versat Architecture. *Electronics* **2021**, *10*, 669. [CrossRef]
11. Papon, C. VexRiscv: A FPGA-Friendly 32-bit RISC-V CPU Implementation. GitHub Repository. 2017. Available online: https://github.com/SpinalHDL/VexRiscv (accessed on 24 August 2026).
12. Ebeling, C.; Cronquist, D.C.; Franklin, P. RaPiD—Reconfigurable Pipelined Datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications (FPL)*; Springer: Berlin/Heidelberg, Germany, 1996; pp. 126–135. [CrossRef]
13. Mei, B.; Vernalde, S.; Verkest, D.; De Man, H.; Lauwereins, R. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL)*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 61–70. [CrossRef]
14. Wijerathne, D.; Li, Z.; Mitra, T. Accelerating Edge AI with Morpher: An Integrated Design, Compilation and Simulation Framework for CGRAs. *arXiv* **2023**, arXiv:2309.06127.
15. Gobieski, G.; Ghosh, S.; Heule, M.; Mowry, T.C.; Nowatzki, T.; Beckmann, N.; Lucia, B. RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*; IEEE: Piscataway, NJ, USA, 2022; pp. 546–564. [CrossRef]
16. Gobieski, G.; Atli, A.O.; Mai, K.; Lucia, B.; Beckmann, N. Snafu: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA '21)*; IEEE: Piscataway, NJ, USA, 2021; pp. 1027–1040. [CrossRef]
17. Carpentieri, N.; Sapriza, J.; Schiavone, D.; Pagliari, D.J.; Atienza, D.; Martina, M.; Burrello, A. Performance Evaluation of Acceleration of Convolutional Layers on OpenEdgeCGRA. *arXiv* **2024**, arXiv:2403.01236. [CrossRef]
18. Flamand, E.; Rossi, D.; Conti, F.; Loi, I.; Pullini, A.; Rotenberg, F.; Benini, L. GAP-8: A RISC-V SoC for AI at the Edge of the IoT. In *Proceedings of the 29th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*; IEEE: Piscataway, NJ, USA, 2018. [CrossRef]
19. Garofalo, A.; Rusci, M.; Conti, F.; Rossi, D.; Benini, L. PULP-NN: Accelerating Quantized Neural Networks on Parallel Ultra-Low-Power RISC-V Processors. *Philos. Trans. R. Soc. A* **2020**, *378*, 20190155. [CrossRef] [PubMed]
20. Machetti, S.; Schiavone, P.D.; Müller, T.C.; Peón-Quirós, M.; Atienza, D. X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators. *arXiv* **2024**, arXiv:2401.05548. [CrossRef]
21. Genc, H.; Kim, S.; Amid, A.; Haj-Ali, A.; Iyer, V.; Prakash, P.; Zhao, J.; Grubb, D.; Liew, H.; Mao, H.; et al. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC '21)*; IEEE: Piscataway, NJ, USA, 2021; pp. 769–774. [CrossRef]
22. Amid, A.; Biancolin, D.; Gonzalez, A.; Grubb, D.; Karandikar, S.; Liew, H.; Magyar, A.; Mao, H.; Ou, A.; Pemberton, N.; et al. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* **2020**, *40*, 10–21. [CrossRef]
23. Abarajithan, G.; Ma, Z.; Munasinghe, R.; Restuccia, F.; Kastner, R. CGRA4ML: A Hardware/Software Framework to Implement Neural Networks for Scientific Edge Computing. *arXiv* **2024**, arXiv:2408.15561.

24. Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18), Carlsbad, CA, USA, 8–10 October 2018; pp. 578–594.
25. Umuroglu, Y.; Fraser, N.J.; Gambardella, G.; Blott, M.; Leong, P.; Jahre, M.; Vissers, K. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*; ACM: New York, NY, USA, 2017; pp. 65–74.
26. Duarte, J.; Han, S.; Harris, P.; Jindariani, S.; Kreinar, E.; Kreis, B.; Ngadiuba, J.; Pierini, M.; Rivera, R.; Tran, N.; et al. Fast inference of deep neural networks in FPGAs for particle physics. *J. Instrum.* **2018**, *13*, P07027. [CrossRef]
27. Alladi, S.; Lopoukhine, A.; Alexandris, G.; Nardi-Dei, A.; Reddy, R.R.; Lamprakos, C.P.; Chaidos, P.; Maras, A.; Ros, A.; Grosser, T.; et al. Optimize edge AI processing through innovative compilation techniques. In *Proceedings of the 2026 Design, Automation & Test in Europe Conference (DATE)*; IEEE: Piscataway, NJ, USA, 2026; pp. 1–7.
28. AMD Inc. *Vitis AI User Guide (UG1414)*; AMD Inc.: Santa Clara, CA, USA, 2024.
29. Moreau, T.; Chen, T.; Vega, L.; Roesch, J.; Yan, E.; Zheng, L.; Fromm, J.; Jiang, Z.; Ceze, L.; Guestrin, C.; et al. A Hardware-Software Blueprint for Flexible Deep Learning Specialization. *IEEE Micro* **2019**, *39*, 8–16. [CrossRef]
30. Takamaeda-Yamazaki, S. NNgen: A Fully-Customizable Hardware Synthesis Compiler for Deep Neural Network. GitHub Repository. 2017. Available online: https://github.com/NNgen/nngen (accessed on 24 August 2026).
31. NVIDIA. NVDLA: NVIDIA Deep Learning Accelerator. Open-Source Project. 2018. Available online: http://nvdla.org (accessed on 24 August 2026).
32. Coreworks SA. SideWorks Reconfigurable DSP Data Engine and FireWorks 32-bit RISC Processor. Design-Reuse Product Announcement. 2008. Available online: https://www.design-reuse.com/news/19500/digital-signal-processing-dsp-core.html (accessed on 24 August 2026).
33. Coreworks SA. First Reconfigurable Licensable Core to Be Independently Benchmarked with the BDTI DSP Kernel Benchmarks. Design-Reuse Press Release. 2009. Available online: https://www.design-reuse.com/news/19919/mid-grain-array-reconfigurable-architecture.html (accessed on 24 August 2026).
34. Mário, V.; Lopes, J.D.; Véstias, M.P.; de Sousa, J.T. Implementing CNNs Using a Linear Array of Full Mesh CGRAs. In *Proceedings of the Applied Reconfigurable Computing (ARC 2020)*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2020; Volume 12083, pp. 288–297. [CrossRef]
35. Miranda, P.R.; Pestana, D.; Lopes, J.D.; Duarte, R.P.; Véstias, M.P.; Neto, H.C.; de Sousa, J.T. Configurable Hardware Core for IoT Object Detection. *Future Internet* **2021**, *13*, 280. [CrossRef]
36. de Sousa, J.T.; Rodrigues, C.A.; Barreiro, N.; Fernandes, J.C. Building Reconfigurable Systems Using Open Source Components. In Proceedings of the X Jornadas Sobre Sistemas Reconfiguráveis (REC 2014), Vilamoura, Portugal, 13 April 2014.
37. OpenRISC Project. OpenRISC 1000 Architecture Manual. 2012. Available online: https://openrisc.io/architecture.html (accessed on 24 August 2026).
38. OpenCores. ORPSoC: OpenRISC Reference Platform System-on-Chip. 2012. Available online: https://opencores.org/projects/orpsoc (accessed on 24 August 2026).
39. Herveille, R. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Revision B.3. OpenCores Organization. 2002. Available online: https://cdn.opencores.org/downloads/wbspec_b3.pdf (accessed on 24 August 2026).
40. Fiolhais, L.; de Sousa, J.T. Warpbird: An Untethered System on Chip Using RISC-V Cores and the Rocket Chip Infrastructure. In Proceedings of the XIV Jornadas sobre Sistemas Reconfiguráveis (REC 2018), Monte da Caparica, Portugal, 15–17 February 2018; pp. 42–49.
41. Asanović, K.; Avizienis, R.; Bachrach, J.; Beamer, S.; Biancolin, D.; Celio, C.; Cook, H.; Dabbelt, D.; Hauser, J.; Izraelevitz, A.; et al. *The Rocket Chip Generator*; Technical Report UCB/EECS-2016-17; EECS Department, University of California: Berkeley, CA, USA, 2016.
42. Bachrach, J.; Vo, H.; Richards, B.; Lee, Y.; Waterman, A.; Avižienis, R.; Wawrzynek, J.; Asanović, K. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Design Automation Conference (DAC)*; ACM: New York, NY, USA, 2012; pp. 1216–1225. [CrossRef]
43. Fiolhais, L.; Gonçalves, F.; Duarte, R.P.; Véstias, M.P.; de Sousa, J.T. Low Energy Heterogeneous Computing with Multiple RISC-V and CGRA Cores. In *Proceedings of the 2019 IEEE International Symposium on Circuits and Systems (ISCAS)*; IEEE: Piscataway, NJ, USA, 2019. [CrossRef]
44. Williams, S. Icarus Verilog. 2002. Available online: https://sourceforge.net/projects/iverilog/ (accessed on 24 August 2026).
45. Snyder, W. Verilator. 2024. Available online: https://www.veripool.org/verilator (accessed on 24 August 2026).
46. Schiavone, P.D.; Conti, F.; Rossi, D.; Gautschi, M.; Pullini, A.; Flamand, E.; Benini, L. Slow and Steady Wins the Race? A Comparison of Ultra-Low-Power RISC-V Cores for Internet-of-Things Applications. In *Proceedings of the 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*; IEEE: Piscataway, NJ, USA, 2017. [CrossRef]
47. IObundle. Versat-AI. 2026. Available online: https://github.com/IObundle/versat-ai (accessed on 24 August 2026).

48. Fazlali, M.; Zakerolhosseini, A.; Gaydadjiev, G. Efficient datapath merging for the overhead reduction of run-time reconfigurable systems. *J. Supercomput.* **2012**, *59*, 636–657. [CrossRef]
49. Carraghan, R.; Pardalos, P.M. An Exact Algorithm for the Maximum Clique Problem. *Oper. Res. Lett.* **1990**, *9*, 375–382. [CrossRef]
50. Tomita, E.; Seki, T. An Efficient Branch-and-Bound Algorithm for Finding a Maximum Clique. In *Proceedings of the Discrete Mathematics and Theoretical Computer Science (DMTCS 2003)*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2731, pp. 278–289. [CrossRef] [PubMed]
51. Prosser, P. Exact Algorithms for Maximum Clique: A Computational Study. *Algorithms* **2012**, *5*, 545–587. [CrossRef]
52. Moon, J.W.; Moser, L. On Cliques in Graphs. *Isr. J. Math.* **1965**, *3*, 23–28. [CrossRef]
53. Microsoft. ONNX Runtime. 2018. Available online: https://onnxruntime.ai (accessed on 24 August 2026).
54. Räiskilä, K. ONNX2C: Open Neural Network Exchange to C Compiler. GitHub Repository. 2020. Available online: https://github.com/kraiskil/onnx2c (accessed on 24 August 2026).
55. Lai, L.; Suda, N.; Chandra, V. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. *arXiv* **2018**, arXiv:1801.06601. [CrossRef]
56. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*; IEEE: Piscataway, NJ, USA, 2016; pp. 770–778. [CrossRef]
57. Krizhevsky, A. *Learning Multiple Layers of Features from Tiny Images*; Technical Report; University of Toronto: Toronto, ON, Canada, 2009.
58. Koizumi, Y.; Saito, S.; Uematsu, H.; Harada, N.; Imoto, K. ToyADMOS: A dataset of miniature-machine operating sounds for anomalous sound detection. In *Proceedings of the 2019 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*; IEEE: Piscataway, NJ, USA, 2019; pp. 313–317.
59. Zhang, Y.; Suda, N.; Lai, L.; Chandra, V. Hello Edge: Keyword Spotting on Microcontrollers. *arXiv* **2018**, arXiv:1711.07128v3. [CrossRef]
60. Warden, P. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv* **2018**, arXiv:1804.03209. [CrossRef]
61. Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv* **2017**, arXiv:1704.04861. [CrossRef]
62. Chowdhery, A.; Warden, P.; Shlens, J.; Howard, A.; Rhodes, R. Visual wake words dataset. *arXiv* **2019**, arXiv:1906.05721.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.