Formal verification of RTL Systolic Arrays for GEMM

EPFL Formal Verification Project

PEDRO PALACIOS ALMENDROS and ADRIAN SCHEMEL, EPFL, Switzerland

1 Introduction

Since their introduction, systolic arrays adoption has been consistently growing for several years as a way to accelerate matrix multiplication in hardware, especially in the machine learning field [10]. As a consequence, formally verifying these components has become more and more fundamental, since a bug in their implementation can be naturally masked by the statistical nature of the applications leveraging them. A well-known example of the importance of hardware verification can be found in the famous `FDIV` bug affecting the floating point unit of some Intel Pentium CPUs [8].

The final goal of our project is to formally verify the RTL implementation of a 2-D systolic array using the SymbiYosys [11] hardware formal verification tool. By leveraging temporal logic, expressed using SystemVerilog Assertions [4], we prove the equivalence between the matrix-matrix multiplication performed on the systolic array module and its mathematical triple-loop definition. Furthermore, we also explore the trade-offs of verifying I/O interfaces of varying complexity and programmability, investigate adding supplementary assertions to accelerate verification, and provide comprehensive benchmarking of all configurations to quantitatively analyze verification performance.

2 Background

In this section, we first introduce the systolic array parallel architecture for computing GEMM (General Matrix Multiplication), then provide a brief introduction to Temporal Logic, which is used to specify correctness and liveness properties of hardware systems. We then present SystemVerilog Assertions, the framework in our chosen Hardware Description Language for specifying time-dependent assertions, and finally describe the formal verification methods used by SymbiYosys, the tool we employed to verify our systolic array hardware implementation.

2.1 Systolic Arrays for GEMM

Systolic arrays are parallel architectures where data flows rhythmically through processing elements arranged in a 2D mesh, with each processing element connected only to its immediate neighbors.

Systolic arrays have found widespread use in matrix-matrix multiplication (GEMM) computations [10] due to their significant advantages: high parallelism (a systolic array with $O(n^2)$ processing elements computes matrix multiplication in $O(n)$ time, compared to the $O(n^3)$ time of the classical triple-loop implementation), efficient data reuse, and a simple hardware design that overcomes the memory bottlenecks inherent to the von Neumann architecture.

In our GEMM systolic array implementation, elements from input matrix A flow horizontally through the grid of PEs, while elements of matrix B remain stationary in the PE registers. Each PE performs multiply-accumulate operations on the incoming data, gradually computing rows of the output matrix C. A hardware diagram of our implemented systolic array can be seen in Figure 1.

To ensure correct alignment of matrix elements for multiply-accumulate operations, the input data must be skewed before entering the array, and the output must be de-skewed accordingly. The systolic array is pipelined to achieve high clock frequencies by registering the data movement between PEs.

Authors' Contact Information: Pedro Palacios Almendros, pedro.palaciosalmendros@epfl.ch; Adrian Schemel, adrian.schemel@epfl.ch, EPFL, Lausanne, Switzerland.

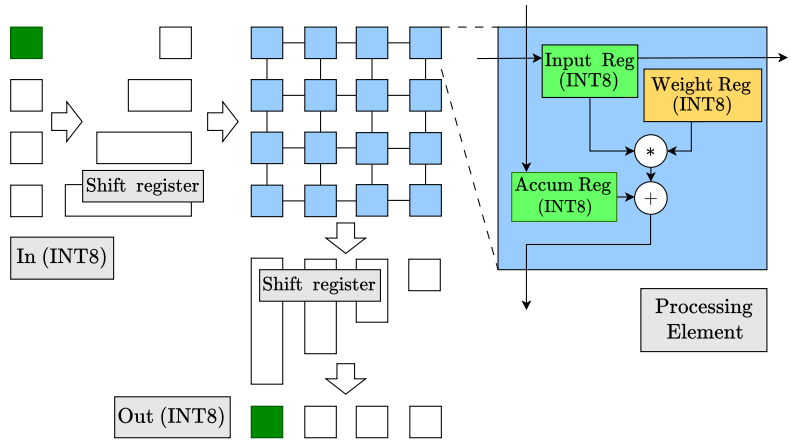

Fig. 1. Hardware diagram of a 4×4 systolic array for matrix-matrix multiplication.

2.2 Temporal Logic

Temporal logic extends boolean logic by introducing primitives that enable the specification of time-dependent properties [5]. It has become a fundamental tool for specifying and verifying properties of hardware systems [1].

While boolean logic can only describe system properties at a single point in time, temporal logic can express properties that hold across the entire state space of a system as it evolves over time. Various formulations of temporal logic exist, but we focus here on the specific temporal logic primitives used in our work, following the notation of [1].

Temporal logic discretizes time into consecutive non-overlapping steps, which in hardware verification naturally correspond to clock cycles. Each step then represents one tick of the hardware clock, at which the system state may change.

2.2.1 The **next** *operator.* The **next** operator is a unary temporal operator specifying that a logical formula will hold in the immediately following time step. For a logical formula A, the expression **next** A means that A will be true in the next time step.

The power of the **next** operator lies in its ability to be composed, allowing us to reason about properties that will hold any number of time steps into the future. We can formally define this through recursive composition:

$$\mathbf{next}^0 A \equiv A \tag{1}$$

$$\mathbf{next}^{n+1} A \equiv \mathbf{next}(\mathbf{next}^n A) \tag{2}$$

This operator forms a fundamental building block for specifying temporal properties. Using **next**, we can construct more complex temporal operators and specifications, as we will see in the following subsections.

2.2.2 The henceforth ($\Box$) operator. This operator expresses that a property is true and will remain true at all subsequent time steps. In the context of verification, this operator is the basis of all "safety" properties, or invariants. It can be used as a global invariant, or also combined with a $\mathbf{next}^n$ operator to delay the start of the check to any future time step, specifying that a certain property must hold perpetually after a certain point in time.

Note that the `henceforth` operator can be defined using a conjunction of **next** operators, due to the discrete nature of time:

$$\Box A \equiv \bigwedge_{i=0}^{\infty} \mathbf{next}^i A \tag{3}$$

2.2.3 The eventually (∇) operator. This operator expresses that a property will be true at some unspecified future moment in time. In hardware verification, this operator is essential for specifying "liveness" properties, properties that assert the system will eventually progress to a desired state.

Analogous to `henceforth`, `eventually` can be defined as a disjunction of **next** operators:

$$\nabla A \equiv \bigvee_{i=0}^{\infty} \mathbf{next}^i A \tag{4}$$

2.3 SystemVerilog Assertions

SystemVerilog, the Hardware Description Language we used to implement our systolic array, provides robust support for time-dependent assertions through its SystemVerilog Assertions (SVA) framework [4, 9]. SVA enables hardware designers to specify temporal logic properties that the implementation must satisfy throughout its operation. These properties can be checked through simulation, or can also be automatically formally verified using tools such as JasperGold [2] or SymbiYosys [11].

2.3.1 SVA Directives. SystemVerilog Assertions introduces three key directives for specifying temporal properties:

```
assume property (sequence_expr);
assert property (sequence_expr);
cover property (sequence_expr);
```

These directives operate on sequence expressions that combine boolean conditions with temporal operators to specify time-dependent behavior. Each directive serves a distinct purpose:

- The `assume` directive specifies environmental constraints and boundary conditions, such as reset signal behaviors, that are assumed to be true by the formal verification tool.
- The `assert` directive defines properties that the design must always satisfy, including both invariant ("safety") conditions and liveness properties. These assertions are the core specifications that will be formally verified using the RTL code.
- The `cover` directive defines specific design states (states that satisfy certain conditions) that must be reachable from the reset state. The formal verification tool checks reachability and generates a trace waveform with the minimal sequence of steps needed to satisfy those conditions. This can greatly help with debugging and easily generating test cases.

2.3.2 SVA Sequence Expressions. The `assert property` and `assume property` directives in SystemVerilog are concurrent assertions, which means that they are evaluated at every rising edge of the clock. Therefore, each concurrent assertion implicitly includes the temporal logic operator $\Box$, meaning the specified property must hold true at all clock ticks during system operation.

The SVA temporal operator `##n` ("n cycles delay") acts as the analog to the temporal logic $\mathbf{next}^n$ operator, and can be chained to construct temporal sequences:

```
assert property (
  A ##1 B ##1 C
);
```

$\equiv \quad \Box\,(A \wedge \mathbf{next}\,(B \wedge \mathbf{next}\,C))$

SVA also provides the overlapping implication operator `|->`, that corresponds to logical implication. When the left-hand sequence matches, the right-hand sequence must follow:

```
assert property (
  A |-> B ##1 C
);
```

$\equiv \quad \Box\,(A \implies (B \wedge \mathbf{next}\,C))$

Additionally, SVA includes the `$past` function as the inverse of the **next** operator, allowing references to values from previous clock cycles. This can significantly improve the readability of certain temporal properties.

All these temporal operators are synchronized to the hardware design's clock, with each time step corresponding to one clock cycle in the actual hardware implementation.

2.3.3 SVA Liveness Properties. To specify liveness properties — assertions that something good must eventually happen — SVA provides the `s_eventually` function. This function corresponds to the temporal logic "eventually" operator (∇) and allows us to express unbounded temporal properties.

```
assert property (
  request |-> s_eventually(response)  ≡  □(Request ⟹ ∇Response)
);
```

2.4 Formal Verification Methods in SymbiYosys

SymbiYosys [11], the formal verification tool that we use, implements two different proving approaches for safety properties based on SAT-solving: Bounded Model Checking (BMC) and k-Induction. Liveness properties require a separate verification approach, which we discuss later in this section.

2.4.1 Bounded Model Checking. BMC formally verifies system properties by smartly checking all sequences up to a fixed bounded length j using SAT-solving techniques. Therefore, it can only formally prove correctness up to inputs of a certain bounded size (in our case, input matrices up to a certain number of rows). It constructs a propositional binary formula T_j representing sequences that could lead to property violations, and then attempts to prove this formula is unsatisfiable using a SAT solver:

$$T_j \equiv Init[\bar{s} := \bar{s}^0] \wedge \left(\bigwedge_{i=0}^{j-1} R_i \right) \wedge E[\bar{s} := \bar{s}^j] \tag{5}$$

Yosys automatically derives the components of this formula: R from the RTL implementation, E from the provided SVA assertions, and $Init$ from both the implementation and SVA assumptions.

The choice of the bound j is very important: too small a value may miss violations that occur later in execution, while too large a value can make verification computationally intractable. For our systolic array implementation, we set $j = 2N + n_{input\_rows}$, where N is the systolic array size and n_{input_rows} is the maximum number of rows of the input matrix to be verified. The additional term $2N$ accounts for the pipeline latency, the time required for data to traverse the systolic array from input to output.

2.4.2 k-Induction. k-Induction can prove properties for unbounded sequences (in our case, input matrices of any number of rows) while remaining computationally feasible. It provides a middle ground between strong induction, which cannot be encoded as a finite SAT formula since it requires checking all states up to n for each value of n, and simple induction, which can only reason about adjacent states, making it insufficient for multi-cycle properties needed when verifying pipelined dataflows such as our systolic array.

The method verifies two conditions using a SAT solver, where $P(i)$ is the formula representing the safety properties to check at time step i:

(1) Base case: $P(0) \wedge \ldots \wedge P(k-1)$
(2) Inductive step: $P(n-k) \wedge \ldots \wedge P(n-1) \implies P(n)$

When checking both conditions, Yosys adds the RTL design's state transition relations between consecutive time steps. These relations come directly from the hardware description and define how signals evolve over time, capturing the circuit's actual behavior.

The choice of parameter k is essential. For our systolic array implementation, we require $k \geq 2(N+1)$, where N is the systolic array size. This minimum value is necessary to relate the systolic

array outputs to their corresponding inputs, accounting for the $2N$ cycles of pipeline latency through the systolic array.

2.4.3 Liveness Properties. Liveness properties require a specialized verification approach and are checked using Yosys's dedicated "`live`" mode, which employs a more restricted set of proof engines like "`suprove`". This verification mode is typically slower compared to standard SAT-based property checking, as we will show in our performance analysis in later sections.

3 Implementation and Analysis

We implemented an INT8 pipelined systolic array following the architecture described in Section 2.1 using SystemVerilog. The RTL code, including all formal verification harnesses and configuration files, is available in our GitHub repository [7].

As we will see in the following sections, the systolic array I/O interface design significantly impacts several aspects of verification: the expression of correctness using SVA, verification speed, and even the provability of correctness using k-Induction. We implemented three interfaces, each offering increased programmability at the cost of verification speed and complexity:

- *Interface 1 (baseline):* Fixed weights that cannot change over time but remain arbitrary (the formal verification tool considers all possible weights). A new input matrix row is provided every cycle with no stalling capability.
- *Interface 2:* Fixed and arbitrary weights, with input stalling capability. A new input row is processed only when the `should_advance_computation` input signal is asserted.
- *Interface 3:* Supports three different commands per clock cycle: `CMD_WRITE_WEIGHTS` to load new weights into the systolic array, `CMD_STREAM` to process a new input matrix row, or `CMD_NONE` to perform no operation.

We tried adding supplementary assertions and invariants derived from the systolic array's mathematical properties to potentially help guide Yosys and the SAT-solver, with the goal of accelerating verification. These intuitive properties sometimes helped achieve faster verification, but sometimes harmed verification speed, as we will see in detail in Section 3.6.

For quantitative analysis and comparison of different configurations, we developed an automatic benchmarking tool that measures verification time using both Bounded Model Checking (BMC) and k-Induction (via Yosys `prove` mode) to verify systolic arrays of varying sizes. The experimental setup is detailed in Section 3.1.

3.1 Experimental setup

We implemented the design in SystemVerilog RTL and specified correctness properties using SystemVerilog Assertions (SVA). For formal verification, we utilized SymbiYosys [11] with an evaluation license that includes the Verific parser for SVA support. SymbiYosys offers three verification modes:

- `bmc`: Performs Bounded Model Checking up to a specified depth.
- `prove`: Uses k-Induction to prove that properties hold for all time steps.
- `live`: Verifies liveness properties, ensuring that desired outcomes eventually occur.

Due to the large number of configurations to evaluate, we developed an automated benchmarking framework. Our testing system consisted of an AMD RYZEN 9 7950X3D processor (16 cores with 2-way SMT) with 64GB of DDR5 memory. Individual verification jobs were constrained to 32GB of memory and a 2-hour runtime limit, after which they were terminated.

SymbiYosys supports multiple SAT solver backends, whose performance varies significantly depending on the hardware design characteristics. Using the SymbiYosys Autotune engine selection tool, we identified Boolector [6] as the optimal solver for our systolic array verification use-case. The `suprove` Yosys backend was used for verifying liveness.

We enforced boundary conditions using `assume` statements in SystemVerilog. Our main assumption forces a reset on the first clock cycle, to have a deterministic initial state:

```
initial assume(!resetn);
```

3.2 Interface 1 (Baseline)

Interface 1 is our baseline configuration, providing the simplest possible interface for the systolic array. It features arbitrary pre-loaded weights that remain fixed throughout operation, modeled using the Yosys `(* anyconst *)` attribute. The interface processes a new input matrix row every clock cycle, with no capability for stalling or pausing the input stream.

3.2.1 Temporal Logic specification of the Systolic Array. In this section, we specify the behavior of the systolic array of size $n \times n$ using temporal logic to gain deeper insight into its operation and to help craft correctness assertions, as well as supplementary assertions, which can guide the SAT-solver and may accelerate verification, as we explore in Section 3.6.

This specification is particularly straightforward in the case of Interface 1, as the systolic array cannot be stalled. Therefore, there is a fixed delay of $2n$ cycles from the input to the output of the systolic array. Specifically, the inputs are fed row by row into the skewing shift-register, and the corresponding result rows emerge from the de-skewing shift-register after exactly $2n$ time steps.

Our specification uses an $n \times n$ matrix $(P_{i,j})$ of Processing Elements, along with two n-dimensional vectors I and O for the systolic array input and output. For correctness analysis, we consider the multiplication $Y = A \cdot B$, where B represents the $n \times n$ weight matrix, A the input matrix, and Y the expected output.

Each Processing Element contains a weight memory W, an additive input In_+, a multiplicative input In_*, and an output Out. The systolic array asserts the signal `output_valid` once O contains valid matrix multiplication results from the first input row I.

We are ready to specify the systolic array behavior using temporal logic expressions[1]:

$$\forall i, j \in \{1, \ldots, n\}.\ \square(W(P_{i,j}) = B_{i,j}) \tag{6}$$

$$\forall i \in \{1, \ldots, n\}.\ \square(In_+(P_{1,i}) = 0) \tag{7}$$

$$\forall i \in \{1, \ldots, n\}.\ \square(I_i = \mathbf{next}^i\ In_*(P_{i,1})) \tag{8}$$

$$\forall i \in \{1, \ldots, n\}.\ \square(Out(P_{n,i}) = \mathbf{next}^{n-i+1}\ O_i) \tag{9}$$

$$\forall i \in \{2, \ldots, n\}, j \in \{1, \ldots, n\}.\ \square(Out(P_{i-1,j}) = \mathbf{next}\ In_+(P_{i,j})) \tag{10}$$

$$\forall i, j \in \{1, \ldots, n\}.\ \square(Out(P_{i,j}) = In_+(P_{i,j}) + In_*(P_{i,j}) \times W(P_{i,j})) \tag{11}$$

Note that these formulas do not need to be specified as SystemVerilog assertions, as they are derived directly from the RTL architecture and are not inputted into SymbiYosys in any way. They are included here solely for reference and completeness, as they are useful for designing correctness assertions. Additionally, note that equations (8), (9), and (10) assume continuous streaming without stalls, while (6) applies only to fixed, pre-loaded weight configurations. Both of these assumptions hold in Interface 1.

3.2.2 Correctness and safety. To verify the systolic array's correctness, we compare its output against a reference implementation based on the triple-loop matrix-matrix multiplication mathematical definition. The SystemVerilog golden reference model is shown in Listing 1. Since the array processes and outputs a single row per cycle, our golden model implements matrix-vector multiplication rather than full matrix-matrix multiplication.

[1]Our use of **next** in these temporal logic expressions is an abuse of notation, as we have only defined it for propositional boolean formulas, not values. Nevertheless, it greatly increases readability, and naturally translates to SVA's `$past` function.

Listing 1. Golden reference model

```
function logic golden_model_matrix_vector_multiply_check (
    input logic[INT_WIDTH-1:0] input_vector[SA_SIZE],
    input logic[INT_WIDTH-1:0] actual_sa_output[SA_SIZE]
);
    logic[INT_WIDTH-1:0] expected[SA_SIZE];
    logic does_match = 1'b1;
    // First compute expected result
    for (int i = 0; i < SA_SIZE; i++) begin
        expected[i] = '0;
        // Compute dot product
        for (int j = 0; j < SA_SIZE; j++) begin
            expected[i] += input_vector[j] * weights[j][i];
        end
    end
    // Now compare expected with actual
    for (int i = 0; i < SA_SIZE; i++) begin
        if (expected[i] != actual_sa_output[i]) begin
            does_match = 1'b0;
        end
    end
    return does_match;
endfunction
```

Using this golden model, we express correctness through an SVA property that compares the systolic array output with the expected output from inputs received $2 \cdot \texttt{SA\_SIZE}$ cycles earlier (the input-to-output delay), as shown in Listing 2.

Listing 2. Correctness assertion for Interface 1

```
assert property (
    output_valid |->
        golden_model_matrix_vector_multiply_check(
            $past(inputs, 2*SA_SIZE), out
        ) == 1'b1
);
```

3.2.3 Liveness. For liveness verification, we initially tried the following property using `s_eventually`:

```
assert property (s_eventually(output_valid));
```

However, verifying properties using `s_eventually` requires SymbiYosys `live` mode, which was tractable only for a 2×2 array within reasonable time constraints. For larger systolic arrays, this liveness verification approach became computationally prohibitive.

Nevertheless, since our design has a fixed, known delay of $2 \cdot \texttt{SA\_SIZE}$ cycles, we can reformulate the liveness property as a bounded property:

```
assert property (!output_valid |-> ##(2*SA_SIZE) output_valid);
```

This reformulation allows verification using the `bmc` and `prove` modes with SAT-solver backends, effectively converting the liveness check into a safety property. As demonstrated in the following section, this approach yields significantly better performance.

3.2.4 Results analysis. We tried verifying the systolic array with different sizes using both `bmc` and `prove`, and the results can be seen in Figure 2. We managed to verify `bmc` up to a systolic array of size 32×32 (taking 28min and 10GB) and `prove` up to a systolic array of size 16×16 (taking 12min

and 29GB). It's evident from Figure 2 that there's a memory usage explosion for `prove`, limiting its applicability to only systolic arrays of smaller sizes.

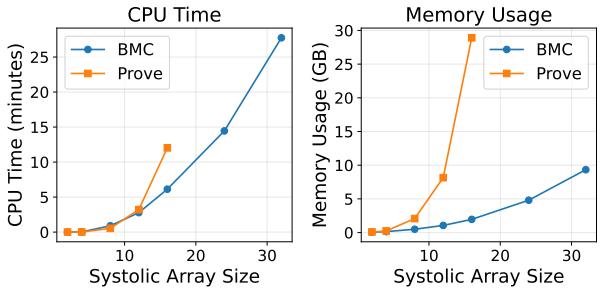

Fig. 2. BMC and Prove performance for Interface 1

Additionally, both the `bmc` and `prove` modes managed to verify much larger systolic arrays than `live`, demonstrating that the reformulation of the liveness property into a bounded property is an effective technique.

3.3 Interface 2

Interface 2 maintains the same fixed weights as Interface 1, but implements a stallable architecture: a new input row is processed only when the `should_advance_computation` input signal is asserted true. When this signal is not asserted, the systolic array's state remains frozen and does not advance. This single change has a profound impact on both expressing correctness properties and the provability using k-Induction, as we will see in the following subsections.

3.3.1 Correctness and liveness properties. Since we no longer have a fixed delay, the verification harness now requires a counter to track the number of "active" cycles (cycles where the `should_advance_computation` signal was asserted high) that have occurred. It also needs to store time-indexed snapshots of the inputs to express correctness assertions: for each output, we must reference the input from $2 \cdot SA\_SIZE$ active cycles ago to compute the expected output using the reference model. This counter also enables specification of the liveness property, as the first output occurs when the counter reaches $2 \cdot SA\_SIZE$. Due to the need to store inputs, we must set a limit on the number of inputs, unlike Interface 1. The correctness and liveness properties are shown in Listing 3.

Listing 3. Interface 2 correction and liveness assertions

```
assert property (output_valid &&
   (should_advance_counter < 2*SA_SIZE + MAX_INPUTS)
       |-> (should_advance_counter >= 2*SA_SIZE)
           && golden_model_matrix_vector_multiply_check(
                should_advance_counter - 2*SA_SIZE,
                out ) == 1'b1);
assert property (
    should_advance_computation &&
    should_advance_counter == 2*SA_SIZE-1 |=> output_valid
);
```

3.3.2 Unprovability using k-Induction. An important consequence of allowing arbitrary-length stalls is that k-Induction becomes impossible: no fixed value of k can be sufficient, as the computation can be delayed for any arbitrarily long period. To understand why, consider that the inductive step in k-Induction must relate inputs to their corresponding outputs. However, for any chosen k, the

systolic array could be stalled for all k cycles, preventing any input from reaching its corresponding output within the induction window. As a result, only Bounded Model Checking (BMC) can be used for verification of Interface 2.

3.3.3 Results analysis. The verification runtime increased dramatically compared to Interface 1, even when using BMC, as shown in Figure 3. BMC completed verification only for a systolic array of size 2×2, taking 13 minutes. For size 4×4, verification was aborted after more than 2 hours.

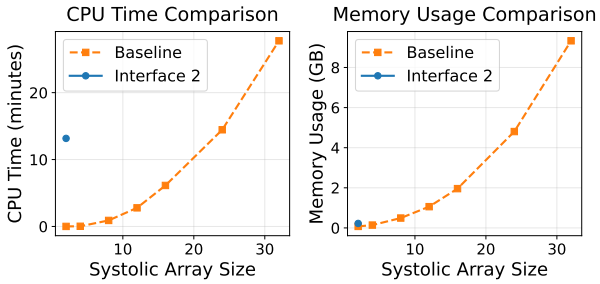

Fig. 3. Interface 2 BMC performance compared with Baseline

3.4 Interface 3

This interface is the most flexible and programmable, supporting one of three commands each cycle: `CMD_WRITE_WEIGHTS` to load new weights into the systolic array, `CMD_STREAM` to process a new input matrix row, or `CMD_NONE` to perform no operation. Given the extremely poor verification performance of Interface 2, which was even simpler than this model, we were forced to verify Interface 3 by driving the hardware model using a simplified state machine test harness rather than supporting arbitrary commands. This simplified state machine loads the weights at initialization and then provides inputs sequentially, one per cycle, mimicking the input behavior of Interface 1.

Therefore, even though the complex hardware module with Interface 3 is used, only input patterns matching the behavior of Interface 1 are formally verified. The rationale for this approach is that simplifying a real-world interface (like Interface 3) to a model that can be efficiently verified (like Interface 1) requires significant HDL modifications. By verifying the actual hardware module, even for a subset of behaviors matching Interface 1, we ensure that the interface simplification process hasn't introduced or removed bugs in these behaviors.

Using the state machine simplifies correctness verification, as we can compute the reference output using the state machine values and compare them with the actual outputs, as shown in Listing 4.

Listing 4. Interface 3 correctness assertion

```
logic match;
always_comb begin
    match = 1'b1;
    for (int i = 0; i < SA_SIZE; i++) begin
        match = match && (activation_outputs[i]
            == reference_outputs[output_row_idx][i]);
    end
end

assert property (
    (output_valid && output_row_idx < INPUT_SIZE) |-> match
);
```

3.4.1 Results analysis. Interestingly, SymbiYosys was unable to complete the inductive step of the `proof` mode for any systolic array size, forcing us to default to BMC. Even with the state machine input pattern simplification, the verification remains extremely slow, as shown in Figure 4, taking more than 15 minutes to verify a 4×4 array.

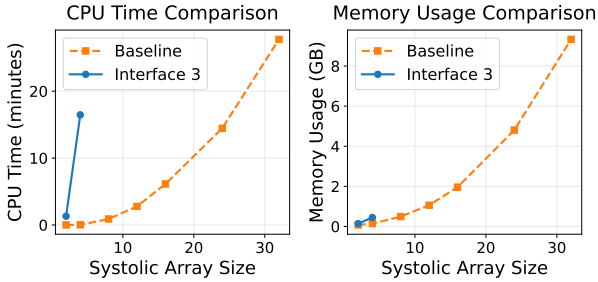

Fig. 4. Interface 3 BMC performance vs Baseline

3.5 Interface Comparison and Summary

As shown in Figure 5, the choice of interface significantly impacts verification speed and even provability using k-induction. Interfaces that allow stalling and deviate from deterministic one-input-per-cycle patterns substantially increase verification complexity, limiting provability to very small systolic arrays.

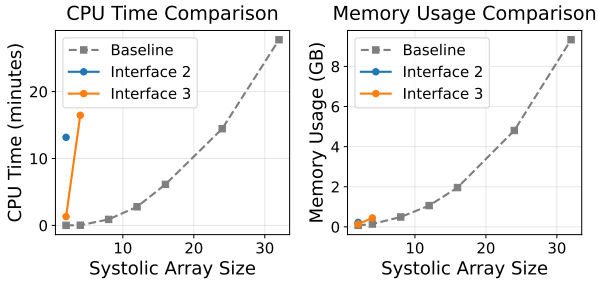

Fig. 5. Interface performance comparison (BMC)

Furthermore, the programmability and complexity of Interface 3 makes verification intractable, even when combined with simple input schemes like one-input-per-cycle using state machine test harnesses. This highlights how a complex interface can negate the benefits of straightforward input patterns that were able to be verified for larger systolic arrays in Interface 1.

Moreover, the unbounded nature of `s_eventually` makes it computationally infeasible for Yosys to prove liveness properties within reasonable time constraints (our experiments could only verify liveness for systolic arrays up to 4×4). Therefore, as we have seen, specifying a precise bounded delay for response completion is preferable when possible (i.e. $\square(\mathit{Request} \implies \mathbf{next}^k\ \mathit{Response})$ for some fixed pre-computed k). While this approach of fixed-delay specification is not universally applicable, it significantly improves verification tractability in the cases it can be used.

On a more positive note, we have been able to verify Interface 1 for non-trivial systolic array sizes (up to 32×32) using Yosys. Industry-grade tools such as JasperGold may allow verifying even larger systolic arrays.

3.6 Supplementary Assertions to Accelerate Verification

Even with Interface 1 (our simplest baseline interface, and the one we will focus on from now on), the verification process is still remarkably slow. Our goal in this section is to explore the addition of supplementary assertions or invariants to the solver. These are not strictly required for correctness, as they can be derived from the systolic array's RTL code, but they may help guide the proof by constraining the state space the SAT solver must explore, possibly accelerating verification.

These supplementary SVA assertions capture mathematical properties of the systolic array that are intuitive to humans but may take the SAT solver significant time to deduce independently.

However, it is important to note that these additional assertions do not always improve verification times. As we will see later in this section, they can sometimes accelerate the verification process, while in other cases, they may slow it down due to the additional burden of proving the supplementary assertions.

3.6.1 Assertion 1: Sub-vector products. This first assertion extends the output correctness check to verify not only the output of the systolic array, but also the partial results of each Processing Element (PE). This assertion translates the mathematical property that, at each time step, the output of each PE is a product of two sub-vectors from the input and weight matrices.

We denote sub-matrices using colon notation [3]. For example, $A(1{:}2,1{:}3)$ represents the sub-matrix of A containing only the top 2 rows and leftmost 3 columns of matrix A. We can express Assertion 1 as follows, where $\langle u, v \rangle$ denotes the dot product between vectors u and v:

$$\forall i, j \in \{1, \ldots, n\}.\ \square \left(\langle I(1{:}i), W(1{:}i, j) \rangle = \mathbf{next}^{i+j-1} Out(P_{i,j}) \right) \tag{12}$$

Notice that the exponent of $\mathbf{next}^{i+j-1}$ depends on i and j, meaning that this property becomes true at different times for different PEs (when all inputs have arrived at that PE and computation is complete), depending on their position within the systolic array.

Its implementation in SVA is shown in Listing 5. Note that the delay argument to `$past` is $i+j+1$, rather than $i+j-1$ as in the temporal logic formula, due to using zero-indexed matrices in SVA as opposed to one-indexed matrices in temporal logic.

Listing 5. Supplementary Assertion 1: Sub-vector products

```
function logic[WEIGHT_ACTIVATION_SIZE-1:0] compute_sub_element(
    input int i, input int j,
    input logic[WEIGHT_ACTIVATION_SIZE-1:0] input_vector[SA_SIZE],
    input logic[WEIGHT_ACTIVATION_SIZE-1:0] weights[SA_SIZE][SA_SIZE]
);  // Computes the dot product of I(1:i) and B(1:i,j)
    logic[WEIGHT_ACTIVATION_SIZE-1:0] result = '0;
    for (int k = 0; k <= i; k++) begin
        result += input_vector[k] * weights[k][j];
    end
    return result;
endfunction

generate
  for (genvar i = 0; i < SA_SIZE - 1; i++) begin
    for (genvar j = 0; j < SA_SIZE; j++) begin
      assert property (
        ##(i+j+2) u_GEMM.u_SA.pe_outs[i][j] ==
          compute_sub_element(i,j,$past(inputs,i+j+1),weights)
      );
    end
  end
endgenerate
```

Unfortunately, adding Assertion 1 results in significantly slower verification times and higher memory usage, as shown in Figure 6. This performance degradation may be caused by the overhead of additional SVA assertions (one per PE) outweighing any reduction in state space search for the SAT solver. This demonstrates that even seemingly reasonable supplementary assertions can sometimes degrade rather than enhance performance.

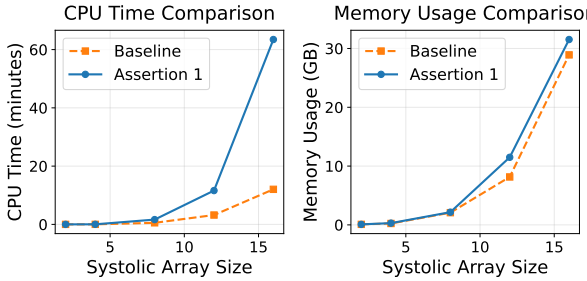

Fig. 6. Assertion 1 performance comparison w.r.t Baseline (k-Induction)

3.6.2 Assertion 2: Input propagation. After Assertion 1's failure to improve verification times, we developed a simpler assertion based on a fundamental property: inputs propagate from left to right over time in the systolic array, meaning that each PE's multiplicative input must correspond to a systolic array input from a specific number of cycles in the past. The rationale behind this supplementary SVA assertion is that explicitly specifying these cycle delays could reduce verification complexity, rather than requiring the solver to derive them through cycle-by-cycle analysis of RTL register chains. The temporal logic formulation of the assertion is shown in Equation 13, and the SVA code in Listing 6.

$$\forall i, j \in \{1, \ldots, n\},\ \square \left(I_i = \mathbf{next}^{i+j-1}\ In_*(P_{i,j})\right) \tag{13}$$

Listing 6. Supplementary Assertion 2

```
generate
    for (genvar i = 0; i < SA_SIZE; i++) begin
        for (genvar j = 0; j < SA_SIZE; j++) begin
            assert property (
                ##(i+j+2) u_GEMM.u_SA.pe_ins[i][j]
                   == $past(inputs[i],i+j+1)
            );
        end
    end
endgenerate
```

This new assertion successfully improved both verification time and memory usage, as shown in Figure 7.

3.6.3 Assertions performance comparison and summary. As shown in Figure 8, carefully crafted assertions can lead to faster verification and reduced memory usage (as demonstrated by Assertion 2). However, this is not guaranteed: other assertions, such as Assertion 1, which seemed promising on paper, may instead harm verification performance. Predicting whether an assertion will help the SAT solver or hinder it by adding additional work is a difficult task. Therefore, the best approach for now appears to be directly implementing the assertions and benchmarking their impact on verification speed.

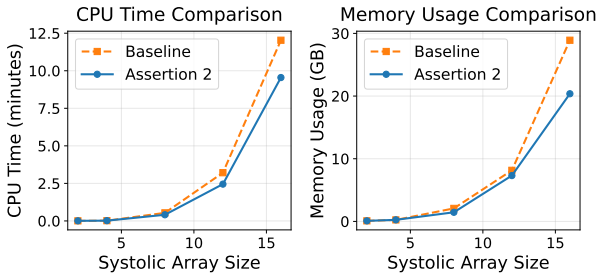

Fig. 7. Assertion 2 performance comparison w.r.t Baseline (k-Induction)

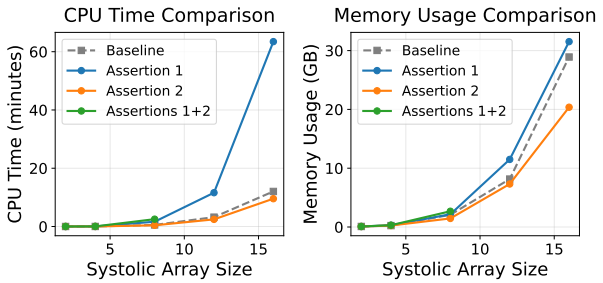

Fig. 8. Assertions performance comparison (k-Induction)

4 Bonus: Using SystemVerilog Assertions as a General-Purpose Constraint Solver

SymbiYosys checks that conditions defined inside `cover` directives are reachable, and generates a waveform trace containing the minimal sequence of steps needed to satisfy those conditions. This is incredibly useful for debugging and test case generation.

Due to generating the shortest waveform of signals that satisfy a condition, `cover` directives can also be used as constraint solvers. SystemVerilog is a very powerful and expressive language, so formal verification tools with SystemVerilog Assertions support can be used for more than just hardware formal verification: they can serve as feature-rich, general-purpose constraint solvers. An algorithm specified in SystemVerilog can be combined with a `cover` statement containing desired constraints on inputs, outputs, and even intermediate values to find valid solutions to the constraint problem.

To explore the possibilities this approach offers, we will use the golden model for matrix-matrix multiplication, as shown in Listing 1, and constrain its inputs or outputs to solve interesting problems using SymbiYosys, such as matrix inversion or LU factorization, specifying only the simple triple-loop GEMM algorithm and `cover` statements in SystemVerilog.

4.1 Forward computation

As a first step, we will constrain the input and weight matrices using a `cover` statement, effectively using the formal verification tool as a simulator to compute the matrix multiplication. That is, we tell SymbiYosys to find output O such that $I \cdot W = O$:

$$\begin{pmatrix} 2 & 5 \\ 3 & 2 \end{pmatrix} \cdot \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 6 & 10 \\ 9 & 4 \end{pmatrix}$$

The result of the `cover` run is a trace containing the result of the multiplication (Fig. 9) which can be loaded in a waveform visualization software like GTKWave.

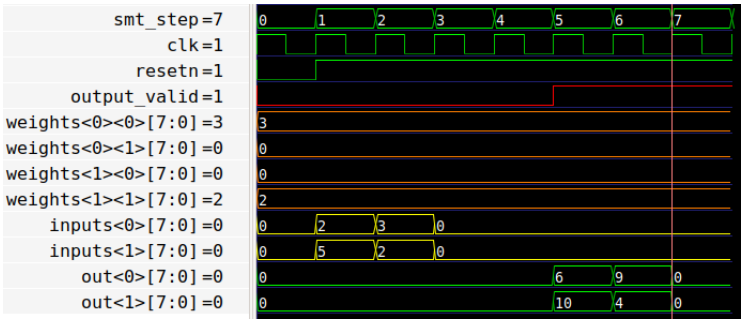

Fig. 9. Output waveform for the matrix-matrix product cover statement

4.2 Backward computation

We can also use `cover` statements in a more interesting way to perform backward computation: by constraining the output in specific ways, we can have the tool determine the necessary inputs and intermediate signals, generating the shortest waveform trace that satisfies these constraints.

For example, by constraining the weight matrix and output matrix, SymbiYosys can perform matrix inversion, as can be seen in Figure 10.

$$\text{Find } I \text{ such that } I \cdot W = O \mod 2^8$$

$$\begin{pmatrix} \texttt{0x4C} & \texttt{0x60} & \texttt{0x6B} \\ \texttt{0xB5} & \texttt{0x86} & \texttt{0xBB} \end{pmatrix} \cdot \begin{pmatrix} 3 & 255 & 1 \\ 4 & 2 & 7 \\ 23 & 42 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \mod 2^8$$

Fig. 10. Matrix inversion using SymbiYosys and SVA cover statements

Furthermore, by constraining the output matrix and partially constraining the input and weight matrices to be lower and upper triangular respectively, SymbiYosys can compute the LU factoring of the output matrix, as shown in Figure 11.

$$\text{Find } L, U \text{ such that } L \cdot U = O \mod 2^8$$

$$\begin{pmatrix} \texttt{0x81} & \texttt{0x00} & \texttt{0x00} \\ \texttt{0x04} & \texttt{0xDD} & \texttt{0x00} \\ \texttt{0x83} & \texttt{0x46} & \texttt{0xD2} \end{pmatrix} \cdot \begin{pmatrix} \texttt{0x81} & \texttt{0x81} & \texttt{0x81} \\ \texttt{0x00} & \texttt{0x8B} & \texttt{0xB7} \\ \texttt{0x00} & \texttt{0x00} & \texttt{0xC3} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 4 & 3 & 255 \\ 3 & 5 & 3 \end{pmatrix}$$

Fig. 11. LU factoring using SymbiYosys and SVA cover statements

5 Conclusion

In this project, we have implemented a parameterizable INT8 systolic array in SystemVerilog, crafted correctness and liveness properties using Temporal Logic and SystemVerilog Assertions, and formally verified the RTL design using SymbiYosys by proving its equivalence to the triple-loop mathematical definition. We have explored several interfaces of varying complexity and programmability and have proposed supplementary assertions to improve verification performance. We have thoroughly benchmarked all configurations to obtain quantitative results and analyzed our findings. Finally, we have demonstrated the potential of using SVA and SymbiYosys as a feature-rich general-purpose constraint solver by computing matrix inversion and LU factorization.

We demonstrated that modern hardware formal verification tools can verify complex designs like systolic arrays up to decent sizes (32×32), provided the interface is constrained to predictable dataflow (without input stalling). We showed that interface choice has a significant impact on both verification speed and provability using k-Induction. We also showed that `s_eventually` liveness properties are considerably slower in SymbiYosys than fixed-delay bounds. Finally, we demonstrated that it is possible, albeit challenging, to improve verification speed and memory usage by adding carefully crafted supplementary assertions.

As future work, it would be valuable to compare the SymbiYosys results with industry-grade tools such as JasperGold, explore additional supplementary assertions to improve verification performance, and analyze systematically which types of supplementary assertions enhance or harm verification speed and memory usage through more rigorous analysis.

References

[1] Bochmann. 1982. Hardware Specification with Temporal Logic: An Example. *IEEE Trans. Comput.* C-31, 3 (1982), 223–231.

[2] Cadence Design Systems. 2024. Jasper Gold Formal Verification Platform. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html

[3] Marcus Hennecke, Ross Moore, and Herb Swan. 2002. https://ccrma.stanford.edu/~jos/matdoc/Rows_columns_submatrices.html

[4] IEEE. 2024. IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)* (2024), 1–1354. https://doi.org/10.1109/IEEESTD.2024.10458102

[5] Zohar Manna and Amir Pnueli. 1979. The modal logic of programs. In *Automata, Languages and Programming*, Hermann A. Maurer (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 385–409.

[6] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014. Boolector 2.0. *J. Satisf. Boolean Model. Comput.* 9, 1 (2014), 53–58. https://doi.org/10.3233/sat190101

[7] Pedro Palacios Almendros and Adrian Schemel. 2025. Formal verification of a GEMM Systolic Array. https://github.com/palmenros/formal_verification_systolic_array_gemm.

[8] D. Price. 1995. Pentium FDIV flaw-lessons learned. *IEEE Micro* 15, 2 (1995), 86–88. https://doi.org/10.1109/40.372360

[9] Srikanth Vijayaraghavan and Meyyappan Ramanathan. 2014. *A Practical Guide for SystemVerilog Assertions.* Springer Publishing Company, Incorporated.

[10] Rui Xu, Sheng Ma, Yang Guo, and Dongsheng Li. 2023. A Survey of Design and Optimization for Systolic Array-based DNN Accelerators. *ACM Comput. Surv.* 56, 1, Article 20 (Aug. 2023), 37 pages. https://doi.org/10.1145/3604802

[11] YosysHQ GmbH. 2024. SymbiYosys. https://symbiyosys.readthedocs.io/en/latest/