Certifying Sequential Consistency of Machine Learning Accelerators

Huan Wu[1], Fei Xie[1], and Zhenkun Yang[2]

[1] Portland State University, Portland, OR 97201, USA
{wuhuan,xie}@pdx.edu
[2] Intel Corporation, Hillsboro, OR 97124, USA
zhenkun.yang@intel.com

Abstract. Machine learning accelerators (MLAs) are increasingly important in many applications such as image and video processing, speech recognition, and natural language processing. To achieve the needed performances and power efficiencies, MLAs are highly concurrent. The correctness of MLAs hinges on the concept of sequential consistency, i.e., the concurrent execution of a program by an MLA must be equivalent to a sequential execution of the program. In this paper, we certify the sequential consistency of modular MLAs using theorem proving. We first provide a formalization of the MLAs and define their sequential consistency. After that, we introduce our certification methodology based on inductive theorem proving. Finally, we demonstrate the feasibility of our approach through the analysis of the NVIDIA Deep Learning Accelerator and the Versatile Tensor Accelerator.

Keywords: Machine Learning Accelerator · Sequential Consistency · Theorem Proving

1 Introduction

Advances in machine learning have led to the widespread adoption of deep learning models in various applications, such as image and video processing, voice recognition, and natural language processing. Existing processors often struggle to meet the computational demands of large-scale machine learning models in terms of training time, inference latency, and power consumption. It has motivated the development of Machine learning accelerators (MLAs) that speed up machine learning in training and inference while lowering power consumption, e.g., Google's Tensor Processing Unit [5], Intel's Nervana Neural Network Processor [13], NVIDIA Deep Learning Accelerator (NVDLA) [11] and Versatile Tensor Accelerator (VTA) [10]. To achieve the required performance and power efficiency, MLAs are highly concurrent and utilize design features such as multi-core and pipelining. However, these concurrent designs may lead to potential issues like race conditions, deadlocks, and non-deterministic outputs. Furthermore, in the MLA ecosystem, the software stack generates sequential workloads that are compiled and executed by hardware. This transition from sequential software execution to concurrent hardware processing potentially introduces data inconsistencies and race conditions.

Therefore, central to the correctness of MLAs is the concept of sequential consistency, that is, a concurrent execution of a program must be equivalent to a sequential execution of the same program. Sequential consistency essentially maps the executions of an MLA to the executions of its sequential reference design. This greatly reduces the complexities in validating the MLA's design. Properties that can be established on the executions of the sequential reference design also hold on to the executions of the concurrent MLA design if the sequential consistency is maintained.

Two major methods are widely used for design validation: simulation-based validation and formal verification. Simulation-based validation exercises the behavior of a design with a series of tests and compares the test results against expectations. However, exhaustive simulation is prohibitively expensive in time and space, and this method only covers a limited set of execution paths, potentially allowing design errors to go undetected. In contrast, formal verification uses a set of formal models, tools, and techniques to mathematically reason about the design and prove its correctness. Theorem proving is a crucial technique in formal verification. It is powerful, imposes no a priori limit on the design size or complexity, and tends to suffer fewer machine-scaling issues than more automated techniques. Nonetheless, it does often require significant human efforts.

In this paper, we present our approach to certifying the sequential equivalence of modular MLAs using inductive theorem proving. Firstly, we propose a formalization of MLAs by formalizing the instruction-driven accelerator design based on the control data flow graph (CDFG). Then, based on this formalization, we prove the sequential consistency of the MLA through induction on the instruction sequence of a program being executed by MLA. Furthermore, we conduct case studies focusing on VTA and NVDLA, demonstrating the feasibility of our approach. Our contributions can be summarized as follows:

1. Formalization of the modular MLAs and their sequential consistency;
2. An inductive theorem proving method to prove the sequential consistency of modular instruction-driven MLAs;
3. Case studies of applying our method to the VTA and NVDLA designs.

The remainder of this paper is organized as follows. Section 2 provides background information on the CDFG, VTA, and NVDLA. Section 3 focuses on the formalization of the modular MLA and its sequential consistency. In Section 4, the proof sketch is presented. The case studies conducted on VTA and NVDLA are discussed in Section 5. Section 6 explores relevant prior work in this area. Finally, Section 7 concludes the study and discusses future work.

2 Background

2.1 Control Data Flow Graph

CDFG combines the concepts of control flow graphs and data flow graphs to model the behavior of a program. Each instruction in a programming language can be decomposed into a series of primitive operations. This set of operations includes assignments, comparisons, arithmetic, logical operations, classic if-then-else, while-loop, and for-loop structures, etc. The control flow represents the sequence of operations performed in a program, organized into basic blocks with distinct entry and exit points. The data flow represents how data is used and modified within a program.

The state of a CDFG is a list of all variables with their corresponding values. To formally define CDFG, let V_{op} be a set of operations involving variables, and V_{bb} be a set of basic blocks, each consisting of a sequence of operations from V_{op}.

Definition 1 (Control Flow and Data Flow Graphs). *A data flow graph is a directed acyclic graph defined as* $G_D \triangleq (V_{op}, E_d)$*, where an edge* $e \in E_d$ *from operation* op_1 *to* op_2 *represents a data dependency of* op_1 *on* op_2*. Similarly, a control flow graph is denoted as* $G_C \triangleq (V_{bb}, E_c)$*, where an edge* $e \in E_c$ *from basic block* bb_0 *to* bb_1 *represents a control dependency of* bb_0 *on* bb_1*.*

Definition 2 (CDFG). *A CDFG is a triple* $G \triangleq (G_D, G_C, R)$*, where* G_D *is the data flow graph,* G_C *is the control flow graph, and* R *is a mapping* $R : V_{op} \rightarrow V_{bb}$ *such that* $R(V_{op_i}) = V_{bbj}$ *if and only if* V_{op_i} *occurs in* V_{bbj}*.*

2.2 Versatile Tensor Accelerator

VTA [10] is an open-source, customizable hardware platform for accelerating tensor-based computations. Figure 1 gives a high-level overview of the VTA architecture. It comprises four modules: fetch, load, compute, and store. Together, these modules define a task pipeline, which enables high compute resource utilization and high memory bandwidth utilization. These modules communicate

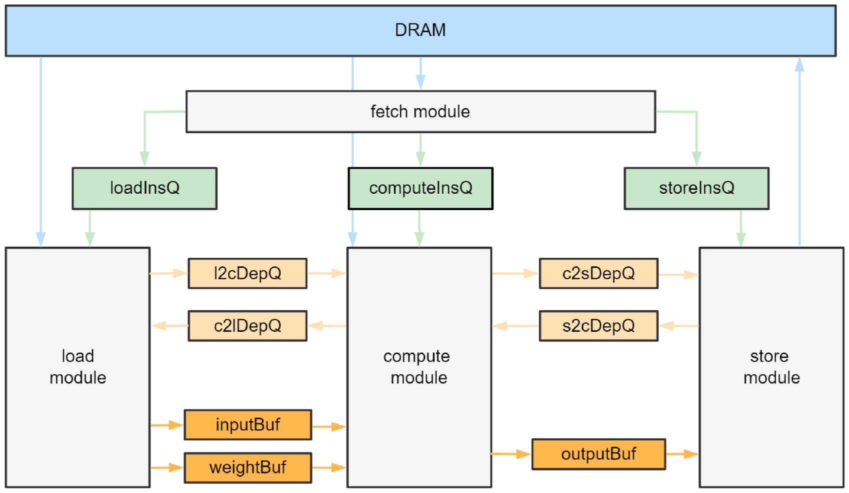


Fig. 1: VTA architecture

via first-in-first-out (FIFO) queues (*l2cDepQ*, *c2lDepQ*, *c2sDepQ*, *s2cDepQ*) and on-chip shared memories (*inputBuf*, *weightBuf*, *outputBuf*) that act as unidirectional data channels. The memory accesses are synchronized through the dependency FIFO queues to prevent data hazards.

2.3 NVIDIA Deep Learning Accelerator

NVDLA is an open-source, scalable deep learning accelerator architecture developed by NVIDIA [11]. It has a modular architecture that can be customized and optimized for specific use cases. The architecture consists of multiple modules working together in a pipeline to perform convolution operations, as shown in Figure 2, including Convolution Direct Memory Access (CDMA), Convolution Buffer (CBUF), Convolution Sequence Controller (CSC), Convolution Multiply-Accumulate (CMAC), and Convolution Accumulator (CACC). Configuration

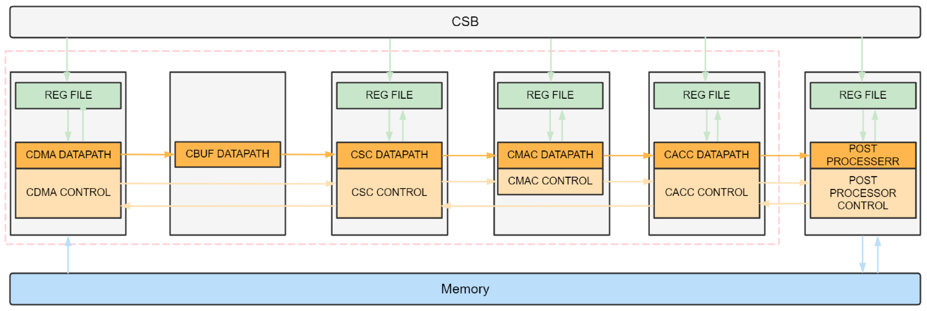


Fig. 2: NVDLA architecture

Space Bus (CSB) serves as an interface that connects the host system with NVDLA.

3 Formalization

To perform verification on an MLA such as VTA, we formalize its design and the desired property into mathematical specifications for theorem proving. MLAs are hardware accelerators designed to speed up machine learning tasks, and they rely on instructions provided by the software to execute these tasks efficiently. These instructions can encompass various aspects, such as defining the model layers and operations to be executed, configuring parameters, and managing data transfers between the host system and the accelerator.

Given the significance of instructions in driving MLA behavior, we propose an instruction-driven architectural pattern (IAP), as depicted in Figure 3. Within the IAP, we define the components involved and establish the necessary constraints. Furthermore, we specify the specific property of sequential consistency.

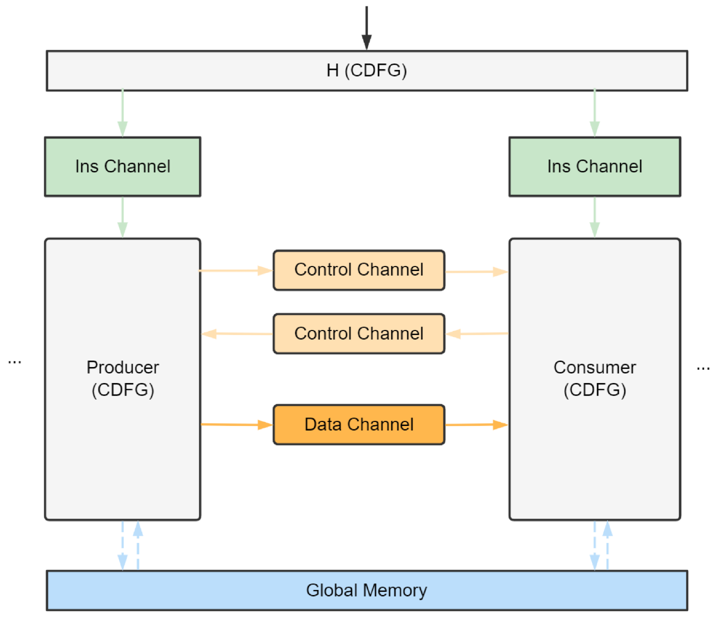


Fig. 3: Instruction-driven Architectural Pattern

The architecture of an MLA is usually composed of multiple modules and global memory. Each module can be viewed as a CDFG. All modules can load and store data in the global memory.

Definition 3 (Global Memory). *Let M be the memory between some CDFGs $G_1, G_2, ..., G_N (N \in \mathbb{N})$, where $G_n(0<n\leq N)$ is capable of either loading data from or storing data to M.*

Data exchange and communication often occur between various modules. To describe these interactions, we introduce the notion of a 'channel'.

Definition 4 (Channel). *A channel, denoted as $c(G_1, G_2)$, represents a dedicated pathway between CDFGs G_1 and G_2 for transmitting data. Each channel is unidirectional, allowing data exchange from source G_1 to destination G_2.*

Channels can be classified based on the transmitted data type: instruction channels for instructions, control channels for control information, and data channels for input and weight data. We now provide a formal definition of the instruction, which is the critical element driving the functionality of MLAs.

Definition 5 (Instruction). *An instruction is defined as $i = \{opcode, memspace, option\}$, where opcode identifies the CDFG to be controlled by this instruction; memspace specifies the address and size of data when the CDFG performs data loading or storing operations; option provides flexibility for accommodating design-specific requirements or custom functionalities.*

Let $(i_n)_{n=1}^{N}$=$\{i_1, i_2, ..., i_N | 1 \leq n \leq N, N \in \mathbb{N}\}$ be an instruction sequence, and s_0 be the initial state of an MLA. We formally define the function *isRunToComplete*$(s_0, (i_n)_{n=1}^{N})$ to determine, starting from s_0, whether all instructions $(i_n)_{n=1}^{N}$ can be executed sequentially and completely, resulting in a final state. The function returns a Boolean value, where the true value indicates that all instructions can be executed completely.

Definition 6 (Valid Instruction Sequence). *A valid instruction sequence is a sequence of instructions $(i_n)_{n=1}^{N}$ satisfying the following conditions:*

1. *The function isRunToComplete$(s_0, (i_n)_{n=1}^{N})$ returns true.*
2. *If there exist i_{k_1} and i_{k_2} $(1 \leq k_1 < k_2 \leq N)$, having the same opcodes and memspace, and no instruction between them has the same opcodes and memspace, and they control the same CDFG to write data in the same space, then there exists i_{k_3} $(k_1 < k_3 < k_2)$ with a different opcode but the same memspace. i_{k_3} controls another CDFG to read the data written by i_{k_1}.*
3. *If multiple instructions that control different CDFGs perform load and store operations on the global memory, their access locations are distinct.*

Now, we formalize the IAP. As depicted in Figure 3 and defined by $\Gamma \triangleq \{(H, G_1, G_2, ..., G_n), \mathcal{C}, M | n \in \mathbb{N}\}$, the IAP encompasses the following components: H, which represents a special CDFG; G_k $(1 \leq k \leq n)$, representing a CDFG, where n denotes the total number of CDFGs excluding H; $\mathcal{C}$, a set of edges representing channels connecting these CDFGs; and M, the global memory. The IAP satisfies the following conditions:

1. Instruction-driven. The CDFG H is responsible for fetching the sequence of instructions $(i_n)_{n=1}^{N}$ from the global memory and distributing them to CDFGs $(G_1, G_2, ..., G_n)$ through the channels $c(H, G_1), c(H, G_2), ..., c(H, G_n)$. Each CDFG G_k $(1 \leq k \leq n)$ executes under the control of these instructions.
2. Producer and consumer pattern. If there exists a data channel between two CDFGs G_1 and G_2, a producer and consumer pattern is established between them. There are two control channels, *p2cCtrlC* and *c2pCtrlC*, and at least a data channel, *dataC*, between G_1 and G_2. An instruction channel, *pInsC*, exists between H and G_1, and another one, *cInsC* exists between H and G_2.

Algorithm 1 Producer(pInsC, p2cCtrlC, c2pCtrlC, dataC)

```
1: var pIns = pInsC.read()
2: var memSpace = pIns.memSpace
3: var readySig = c2pCtrlC.read()
4: while isFalse(readySig) do
5:     skip
   dataC.write(data,memSpace)
6: p2cCtrlC.write(validSig)
```

Algorithm 2 Consumer(cInsC, p2cCtrlC, c2pCtrlC, dataC)

```
1: var cIns = cInsC.read()
2: var memSpace = cIns.memSpace
3: var validSig = p2cCtrlC.read()
4: while isFalse(validSig) do
5:     skip
6: dataC.read(data,memSpace)
7: c2pCtrlC.write(readySig)
```

Algorithm 1 demonstrates the producer mode. Initially, an instruction *pIns* is read from *pInsC*. To produce and transmit new data to the consumer via *dataC*, it is crucial to check if there is available space for production. This is determined by reading a ready signal from *c2pCtrlC*, indicating the consumer's readiness to receive new data. If the consumer is ready to receive the new data, the producer writes data to *dataC* and sets a valid signal in *p2cCtrlC* to inform the consumer of the availability of consumable data. If the consumer is not ready, the producer waits until space becomes available for production.

Algorithm 2 illustrates the consumer mode. Initially, an instruction *cIns* is read from *cInsC*. To consume data through *dataC*, it is necessary to check if the producer has produced data in this space. This is determined by reading a valid signal from *p2cCtrlC*, indicating the presence of new data from the producer. If new data is available, the consumer reads the data from *dataC* and sets a ready signal into *c2pCtrlC* to notify the producer of its readiness to receive new data. If the producer has not yet produced new data, the consumer waits until the data becomes available for consumption.

The IAP has two execution semantics: sequential and concurrent.

Definition 7 (Sequential Semantics). *The instruction sequence $(i_n)_{n=1}^{N}$ distributed by CDFG H is executed in the exact order of $(i_n)_{n=1}^{N}$. Each step involves the execution of a single instruction.*

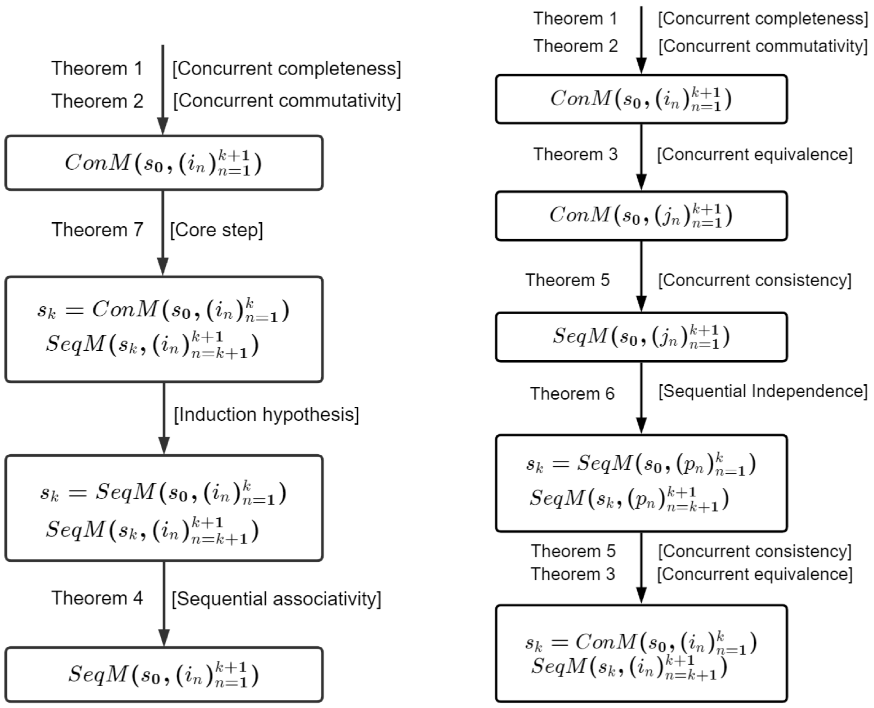
Definition 8 (Concurrent Semantics). *The instruction sequence $(i_n)_{n=1}^{N}$ distributed by CDFG H is executed concurrently, allowing only those instructions that have no dependencies on each other to be executed concurrently in a single step.*

The state of IAP includes the content of the global memory M, channels $\mathcal{C}$, and the state of all CDFGs $H, G_1, G_2, ..., G_n$. We define $SeqM(s, (i_n)_{n=1}^{N})$ and $ConM(s, (i_n)_{n=1}^{N})$ as the state of the IAP obtained from the initial state s after executing the instruction sequence $(i_n)_{n=1}^{N}$ sequentially and concurrently, respectively. Now we specify the property of sequential consistency:

Definition 9 (Sequential Consistency). *Given a valid instruction sequence $(i_n)_{n=1}^{N}$, the initial state s_0, and the IAP Γ, $ConM(s_0, (i_n)_{n=1}^{N})$ is equivalent to $SeqM(s_0, (i_n)_{n=1}^{N})$.*

4 Proof Sketch

We use an induction based on the instruction sequence to prove sequential consistency. In conjunction with the formalization presented in the previous section, we introduce seven auxiliary theorems that are integral to our proof. Figure 4 shows the relationship between these theorems and their role in establishing sequential consistency. Figure 4(a) depicts that sequential consistency is established based on the core step. This core step is to prove that the state obtained after executing a valid instruction sequence $(i_n)_{n=1}^{k+1}$ concurrently is equivalent to the state obtained by executing the first k instructions concurrently and then executing the last instruction i_{k+1}. The proof process of this core step is detailed in Figure 4(b).


Theorem 1 [Concurrent completeness]
Theorem 2 [Concurrent commutativity]
$ConM(s_0, (i_n)_{n=1}^{k+1})$
Theorem 7 [Core step]
$s_k = ConM(s_0, (i_n)_{n=1}^{k})$
$SeqM(s_k, (i_n)_{n=k+1}^{k+1})$
[Induction hypothesis]
$s_k = SeqM(s_0, (i_n)_{n=1}^{k})$
$SeqM(s_k, (i_n)_{n=k+1}^{k+1})$
Theorem 4 [Sequential associativity]
$SeqM(s_0, (i_n)_{n=1}^{k+1})$
Theorem 1 [Concurrent completeness]
Theorem 2 [Concurrent commutativity]
$ConM(s_0, (i_n)_{n=1}^{k+1})$
Theorem 3 [Concurrent equivalence]
$ConM(s_0, (j_n)_{n=1}^{k+1})$
Theorem 5 [Concurrent consistency]
$SeqM(s_0, (j_n)_{n=1}^{k+1})$
Theorem 6 [Sequential Independence]
$s_k = SeqM(s_0, (p_n)_{n=1}^{k})$
$SeqM(s_k, (p_n)_{n=k+1}^{k+1})$
Theorem 5 [Concurrent consistency]
Theorem 3 [Concurrent equivalence]
$s_k = ConM(s_0, (i_n)_{n=1}^{k})$
$SeqM(s_k, (i_n)_{n=k+1}^{k+1})$

(a) Sequential consistency (b) Core step

Fig. 4: Proof Overview

To prove the core step, we introduce the following six theorems. (1) *Concurrent completeness.* A valid instruction sequence can be executed concurrently and completely, reaching a reachable final state. Otherwise, there may be deadlocks. (2) *Concurrent commutativity.* The instructions in each concurrent step can be executed in any order without affecting the final state of each step. Concurrency improves performance while preserving correctness. Concurrent completeness and commutativity are the fundamental properties of concurrent execution and are the premise of other theorems. (3) *Concurrent equivalence.* The state resulting from executing a valid instruction sequence concurrently is consistent with the state obtained from concurrently executing the corresponding instruction sequence rearranged in the concurrent step order of execution. Although the instruction order may differ, the final state remains the same. (4) *Sequential associativity.* The state obtained from sequentially executing all instructions is equivalent to executing the preceding instructions sequentially and then the remaining instructions sequentially. It allows instructions to be grouped or associated differently without affecting the final state. (5) *Concurrent consistency.* The state obtained by concurrently executing the instruction sequence, sorted according to the order of concurrent execution steps, is equal to the state obtained from sequentially executing the same instruction sequence. (6) *Sequential independence.* When an instruction has no dependencies with the other instructions within an instruction sequence, executing this instruction sequentially, followed by the remaining instructions, yields the same state as executing the other instructions first and then executing this instruction.

We present the formulation of the theorems above and provide sketches of the proof for the core step and the final theorem that establishes sequential consistency. Assume the total number of steps required to concurrently execute the instructions $(i_n)_{n=1}^N$ is denoted as T, the sequence of steps is represented by $(t_n)_{n=1}^T$, and the instructions in each step are denoted as $(r_n^t)_{n=1}^R$, where $0<R\leq N$ represents the number of instructions in the step, and $0<t\leq T$ represents the step index. According to the valid instruction sequence definition, $(i_n)_{n=1}^N$ can be executed concurrently and completely, resulting in a reachable final state.

Theorem 1 (Concurrent completeness). *If $isRunToComplete(s_0, (i_n)_{n=1}^N)$ is true, then $s_f = ConM(s_0, (i_n)_{n=1}^N))$, $(i_n)_{n=1}^N$ can be executed completely and s_f is reachable.*

We define the function $isValid(s_0, (i_n)_{n=1}^N)$ to determine whether the sequence of instructions starting from the initial state s_0 is a valid instruction sequence. The function returns a Boolean value, where the true value indicates that $(i_n)_{n=1}^N$ is a valid instruction sequence from s_0.

Theorem 2 (Concurrent Commutativity). *$\forall 0<t\leq T$, let $(a_n^t)_{n=1}^R$ be the instruction sequence with random order of $(r_n^t)_{n=1}^R$. If $isValid(s, (r_n^t)_{n=1}^R)$ is true, then $ConM(s, (r_n^t)_{n=1}^R)) = ConM(s, (a_n^t)_{n=1}^R))$.*

Let the instruction sequence $(j_n)_{n=1}^N$ be the collection of concurrently executed instructions $(i_n)_{n=1}^N$ arranged in the order of steps. By the concurrent semantics, the concurrent equivalence theorem follows.

Theorem 3 (Concurrent Equivalence). *If $isValid(s, (i_n)_{n=1}^N)$ is true, then $ConM(s, (i_n)_{n=1}^N)) = ConM(s, (j_n)_{n=1}^N)$.*

Based on the sequential semantics, if there are N instructions, executing N instructions sequentially results in a state consistent with executing the first A instructions ($A<N$) sequentially and then executing the remaining instructions sequentially.

Theorem 4 (Sequential Associativity). *If $A\leq N$ and $isValid(s, (i_n)_{n=1}^N)$ is true, then $s_a = SeqM(s, (i_n)_{n=1}^A)$, $SeqM(s, (i_n)_{n=1}^N) = SeqM(s_a, (i_n)_{n=A+1}^N)$.*

The concurrent execution of $(j_n)_{n=1}^N$ can be viewed as the sequential execution of the instructions within each concurrent step.

Theorem 5 (Concurrent Consistency). *If $isValid(s,(i_n)_{n=1}^{N})$ is true, then $ConM(s,(j_n)_{n=1}^{N})) = SeqM(s,(j_n)_{n=1}^{N})$.*

To determine whether there is a dependency relationship between instruction i_{k_1} and all instructions in $(i_n)_{n=k_2}^{k_3}$, we use the function $dep(i_{k_1},(i_n)_{n=k_2}^{k_3})$. Here, $k_1{\neq}k_2$, $k_1{\neq}k_3$, and $k_2{\leq}k_3$. The function returns a Boolean value. If the return value is false, it indicates that i_{k_1} and $(i_n)_{n=k_2}^{k_3}$ can run simultaneously without causing a deadlock, thereby implying that i_{k_1} has no dependencies with any of the instructions in $(i_n)_{n=k_2}^{k_3}$. If the first instruction i_1 has no dependence on the remaining instructions in $(i_n)_{n=2}^{N}$, then i_1 can be scheduled to be executed in the last step without changing the final state obtained by executing $(i_n)_{n=1}^{N}$ sequentially.

Theorem 6 (Sequential Independence). *If $N{>}1$, $isValid(s,(i_n)_{n=1}^{N})$ is true and $dep(i_1,(i_n)_{n=2}^{N})$ is false, then $s_t{=}SeqM(s,(i_n)_{n=2}^{N})$, $SeqM(s,(i_n)_{n=1}^{N}) = SeqM(s_t,(i_n)_{n=1}^{1})$.*

Based on the previous theorems, we prove the core step theorem:

Theorem 7 (Core Step). *If $isValid(s,(i_n)_{n=1}^{N+1})$ is true, then $s_1 = ConM(s,(i_n)_{n=1}^{N})$, $ConM(s,(i_n)_{n=1}^{N+1})) = SeqM(s_1,(i_n)_{n=N+1}^{N+1})$.*

Proof. As shown in Figure 4(b), based on Theorem 3 (concurrent equivalence), the original instruction sequence $(i_n)_{n=1}^{N+1}$ can be converted to an instruction sequence $(j_n)_{n=1}^{N+1}$ arranged according to the step order of concurrent execution while preserving the final state. Then according to Theorem 5 (concurrent consistency), where $ConM(s,(j_n)_{n=1}^{N+1}) = SeqM(s,(j_n)_{n=1}^{N+1})$, the concurrent execution can be substituted with the sequential execution. Next, let's consider the last instruction i_{N+1} in $(i_n)_{n=1}^{N+1}$, which is also present in $(j_n)_{n=1}^{N+1}$. Assume $(p_n)_{n=1}^{N}$ is the instruction sequence that preserves the order of the remaining instructions after removing i_{N+1} from $(j_n)_{n=1}^{N+1}$, with $p_{N+1} = i_{N+1}$. Applying Theorem 6 (sequential independence), the sequential execution can be divided into two parts: executing $(p_n)_{n=1}^{N}$ sequentially and then executing p_{N+1}. Additionally, due to Theorem 5 and Theorem 3, the state obtained from executing $(p_n)_{n=1}^{N}$ sequentially is the same as that obtained by executing $(i_n)_{n=1}^{N}$ concurrently.

Finally, we prove the theorem of sequential consistency:

Theorem 8. *if $isValid(s,(i_n)_{n=1}^{N})$ is true, then*
$ConM(s,(i_n)_{n=1}^{N}) = SeqM(s,(i_n)_{n=1}^{N})$.

Proof. As shown in Figure 4(a), by induction on the sequence of instructions.

Base case: if $N = 1$, it is true trivially.

Inductive case: assume $ConM(s,(i_n)_{n=1}^{k}){=}SeqM(s,(i_n)_{n=1}^{k})$ holds, we need to prove $ConM(s,(i_n)_{n=1}^{k+1}){=}SeqM(s,(i_n)_{n=1}^{k+1})$. Let $s_1 = ConM(s,(j_n)_{n=1}^{k})$, since

$ConM(s,(i_n)_{n=1}^{k+1})) = SeqM(s_1,(i_n)_{n=k+1}^{k+1})$ (Theorem 4.8),

$SeqM(s_0,(i_n)_{n=1}^{k+1}) = SeqM(s_1(i_n)_{n=k+1}^{k+1})$ (Theorem 4.5)

Therefore, this theorem holds.

5 Case Studies

For case studies, we utilize VTA and NVDLA to illustrate how these accelerators align with our previously formalized IAP architectural pattern. When the architectures of MLAs adhere to IAP, the proof process outlined in Section 4 can be applied to establish the sequential consistency of the MLA. Our analysis primarily evaluates how well these accelerators adhere to the specifications and characteristics defined in our formalization.

5.1 Case Study 1: VTA

The VTA architecture is an instruction-driven architecture. Each instruction in VTA is encoded with specific fields to indicate the type of operation, control flags, memory addresses, data sizes, and other relevant information. The VTA architecture consists of four functional modules shown in Figure 1, each designed to handle specific tasks. The fetch module, represented by CDFG H in IAP, retrieves an instruction stream from DRAM and decodes the instructions. It routes the instructions to one of three instruction FIFO queues. Within VTA, there are two sets of producer and consumer models, and between each set, there exist two control FIFO queues and at least a data buffer.

The load and compute modules follow the producer and consumer pattern. There are two control FIFO queues *l2cDepQ* and *c2lDepQ*, and two buffers *inputBuf* and *weightBuf* between the load and compute modules. The data and control communication process between these modules is as follows:

– The load module operates by reading an instruction from the load instruction FIFO queue. Each load instruction contains flags associated with control FIFO queues, indicating dependencies on the compute module. If the flag corresponding to *c2lDepQ* is set, the load module checks the status of *c2lDepQ* to determine if it's empty. In the event that *c2lDepQ* is empty, indicating that the consumer is not ready to receive new data, the load module waits for the compute module to write the control information into *c2lDepQ*. Otherwise, the load module proceeds to load input or weight tensors from DRAM into *weightBuf* or *inputBuf*. Additionally, if the flag corresponding to the control queue *l2cDepQ* is set, the load module writes control information into *l2cDepQ*.
– The compute module reads an instruction from the compute instruction FIFO queue. Each compute instruction has flags associated with control FIFO queues, indicating dependencies on the load module. If the flag for *l2cDepQ* is set, the compute module checks the status of *l2cDepQ*. If *l2cDepQ* is found to be empty, indicating that there is no new data available for consumption, the compute module waits for the load module to write the control information into *l2cDepQ*. Otherwise, the compute module proceeds to read data from buffers and performs various computations on the input data. Furthermore, if the flag corresponding to the control queue *c2lDepQ* is set, the compute module writes control information into *c2lDepQ*.

The compute and store models also function following the producer and consumer pattern. Similarly, they have two control FIFO queues, *c2sDepQ* and *s2cDepQ*, and a buffer, *outputBuf*, between them. Similar to the communication process described above between the load and compute modules, both the compute and store modules follow a similar procedure. They start by reading an instruction from their respective instruction queues and then check the corresponding control queue based on the flag specified in the instruction. The compute module stores the computed data in the buffer *outputBuf*, while the store module reads data from *outputBuf* and stores it in DRAM. Finally, they write control information to each control queue based on the flag in the instruction.

The VTA architecture and IAP demonstrate a strong alignment in terms of their instruction-driven nature and the communication between different models. In both cases, there is a module for acquiring instructions and effectively distributing them to their respective instruction channels. Additionally, the communication patterns within VTA exhibit a clear producer-consumer relationship, where data flows from one module to another in a coordinated manner. This correspondence further solidifies the compatibility between the VTA architecture and IAP, reinforcing the effectiveness and accuracy of the formalized model in capturing the essential aspects of the accelerator architecture.

Mechanized Proof In Dafny We use Dafny [9] as our theorem prover to certify the sequential consistency of VTA. Table 1 summarizes the statistics about our Dafny implementation. The "Formalization" column shows the lines of code, including the formalization of the instruction definition, the valid instruction sequence, VTA, and sequential consistency. The "Proof" column shows lines of code of all proofs we need to certify the sequential consistency. The overall verification time for proof-checking all Dafny code is about 35 minutes.

Table 1: Code size and verification time

Formalization (LoC)	Proof (LoC)	Verification Time (Min)
1788	14274	35

We illustrate the implementation details with the theorem that ultimately proves sequential consistency. Figure 5 shows the proof of Theorem 8 in Dafny. The input of lemma function *theorem8* includes: the instruction sequence *insSeq1*; the global memory *gsmem*; the buffers *InputB*, *WeightB*, and *OutputB*; and FIFO queues *L2CQ*, *C2LQ*, *C2SQ*, and *S2CQ*. The precondition requires a valid instruction sequence, denoted as *validInsSeq*, while the property to be proven is sequential consistency, represented as *seqConsistency*. The proof follows an induction method based on the input sequence of instructions. The base case involves only one instruction, and the property holds trivially. In the inductive step, sequential consistency is proved by leveraging *theorem 7*, the induction hypothesis *theorem8*, and *theorem4*. Overall, Theorem 8 is certified by Dafny.

```
lemma theorem8(insSeq1: seq<ins>, gsmem:seq<nat>, InputB: seq<int>,
    WeightB: seq<int>, OutputB: seq<int>, L2CQ:seq<nat>, C2LQ:seq<nat>,
    C2SQ:seq<nat>, S2CQ:seq<nat>)
  requires validInsSeq(insSeq1, L2CQ, C2LQ, C2SQ, S2CQ)
  ensures seqConsistency(insSeq1, gsmem, InputB, WeightB, OutputB,
              L2CQ, C2LQ, C2SQ, S2CQ)
{ var length := |insSeq1|;
  if length <= 1
  { assert true;
  }else{
    assert seqConsistency(insSeq1, gsmem, InputB, WeightB, OutputB,
        L2CQ, C2LQ, C2SQ, S2CQ) by {
      theorem7(insSeq1, gsmem, InputB, WeightB, OutputB,
        L2CQ, C2LQ, C2SQ, S2CQ);
      theorem8(insSeq1[..length-1], gsmem, InputB, WeightB, OutputB,
        L2CQ, C2LQ, C2SQ, S2CQ);
      theorem4(insSeq1, gsmem, InputB, WeightB, OutputB,
        L2CQ, C2LQ, C2SQ, S2CQ);}
  }}
```

Fig. 5: Mechanized Proof of Theorem 8 in Dafny

5.2 Case Study 2: NVDLA

NVDLA also follows an instruction-driven architecture. The CSB module facilitates communication between the host system and NVDLA, allowing the host system to send commands and configuration parameters to define the behavior and settings of NVDLA. CSB acts as the CDFG H in IAP and distributes instructions to the register file in various modules within NVDLA. These instructions can include configuration parameters, control commands, memory addresses, and other information for configuring and controlling the accelerator. The specific instructions that CSB distributes depend on the desired operation and functionality of NVDLA, as specified by the host system.

The NVDLA convolution core pipeline consists of 5 stages that work together to perform convolution operations efficiently. The CDMA is responsible for fetching input and weights from memory and storing the data in CBUF, which acts as a buffer for holding the received data. The CSC controls the sequencing of convolution operations. It takes input and weights from CBUF and distributes them to the relevant CMAC units for processing. The CMAC performs the convolutions, receiving CSC data and executing the multiply and accumulate operations. The CACC accumulates the results from CMAC by collecting the partial products generated and combining them to produce the final output. There are two sets of producers and consumers.

The CDMA and CSC follow the producer and consumer pattern. There are two ports, $sc2cdmaC$ and $cdma2scC$, and a buffer CBUF between CDMA and

CSC. These ports facilitate the transmission of CBUF's status between CDMA and CSC. The communication process follows these steps:

- The CDMA reads the instruction from the register file to determine the data and weights to be fetched from memory. It checks the status of the CBUF using port *sc2cdmaC* to determine if there is available space in the CBUF to store the data. If space is available, the CDMA writes the data into the CBUF and sends the current status of the CBUF to the CSC through port *cdma2scC*, informing the CSC about the data availability. If there is no space, the CDMA waits until space becomes available.
- The CSC reads the instruction from the register file to determine which data to retrieve from the CBUF. It checks the status of the CBUF using the port *cdma2scC* to determine if there is data available in CBUF for processing. If data is available in the CBUF, the CSC reads the data from the CBUF for further processing and sends the updated status of the CBUF to the CDMA through the port *sc2cdmaC*. This status update informs the CDMA about the current status of the CBUF after data retrieval. If no data is available, the CSC waits until data becomes available.

The CSC, CMAC, and CACC also follow the producer and consumer pattern. The CSC serves as the producer model, while the combined CMAC and CACC modules function as the consumer model. There is a data port *sc2macDC* and a control port *sc2macC* between the CSC and the CMAC. Similarly, there is a data port *mac2accDC* and a control port *mac2accC* between the CMAC and the CACC. Additionally, there is a control port *acc2scC* between CACC and CSC. The communication process follows these steps:

- The CSC reads the instruction from the register file. It checks the credit signal from the CACC through *acc2scC* to determine if there is available space for the CACC to perform computations. If space is available, the CSC sends the data to the CMAC through *sc2macDC* and sends the valid signal to the CMAC through *sc2macC*. If no space is available, the CSC waits until space becomes available.
- The CMAC reads the instruction from the register file. It checks the valid signal from CSC through *sc2macC* to determine if there is valid data to receive. If there is valid data, the CMAC gets the data and performs the convolution computation, producing intermediate results. The intermediate data is then sent to the CACC through *mac2accDC*, and the valid signal is sent to the CACC through *mac2accC*. If there is no valid data, the CMAC waits until valid data is available.
- The CACC reads the instruction from the register file. It checks the valid signal from CMAC through *mac2accC* to determine if there is valid data to receive. If there is valid data, the CACC gets the data and performs the accumulated operations. The CACC sends a credit signal to the CSC through *acc2scC*, indicating the space available for the CSC. If there is no valid data, the CACC waits until valid data is available.

The NVDLA architecture aligns with IAP. The CSB plays the role of CDFG H in the pattern. Within NVDLA, there are two sets of producers and consumers. In one set, the CDMA is the producer, while the CSC is the consumer. In the other set, the CSC is the producer, and the CMAC and CACC act as consumers. Using Theorem 7, Theorem 4, and the induction method, we can establish the sequential consistency of NVDLA.

6 Related Work

There have been many approaches to certifying concurrent processor features using theorem proving techniques. For example, Kroening et al.[8] demonstrate the correctness of generating a pipelined microprocessor from an arbitrary sequential specification. They employ the PVS proof assistant [2] to implement this proof. Sawada et al.[14] verify the equivalent of the state transitions of pipelined and non-pipelined machines in the presence of external interrupts. They create a table-based model of pipeline execution and achieve this proof in the ACL2 theorem prover[6]. Damm et al.[3] establish the property that out-of-order execution produces the same final state as a purely sequential machine running the same program. Their proof is based on the semantic model of synchronous transition systems[12]. Vijayaraghavan et al.[15] develop a modular proof structure to prove that the distributed shared-memory hardware system implements sequential consistency. This method is based on labeled transition systems (LTSes) theory [7], and the proof is carried out using the Coq proof assistant[1].

The statement of correctness in our work is sequential consistency; that is, the MLA produces the same final state as the same design with sequential semantics. The formalization follows the style of Communicating Sequential Processes (CSP) [4] and adds features to formalize MLA designs.

7 Conclusions and Future Work

This paper presents a comprehensive formalization of MLAs and specifies and certifies their sequential consistency, that is, the concurrent execution of a program by the MLA is equivalent to a sequential execution of the program by its sequential reference design. This finding is crucial as it paves the way for simplifying the verification process of concurrent MLAs by leveraging their sequential counterparts. Building upon the foundation of sequential consistency, in future work, we can explore and validate various properties of concurrent MLAs, such as correctness, optimizations, resource utilization, or novel execution models.

Acknowledgement

This research is partially supported by a gift from Intel Corporation.

References

1. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media (2013)
2. Cyrluk, D., Rajan, S., Shankar, N., Srivas, M.K.: Effective theorem proving for hardware verification. In: Theorem Provers in Circuit Design: Theory, Practice and Experience Second International Conference, TPCD'94 Bad Herrenalb, Germany, September 26–28, 1994 Proceedings. pp. 203–222. Springer (1995)
3. Damm, W., Pnueli, A.: Verifying out-of-order executions. In: Advances in Hardware Design and Verification: IFIP TC10 WG10. 5 International Conference on Correct Hardware and Verification Methods, 16–18 October 1997, Montreal, Canada. pp. 23–47. Springer (1997)
4. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8), 666–677 (1978)
5. Jouppi, N.P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al.: In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th annual international symposium on computer architecture. pp. 1–12 (2017)
6. Kaufmann, M., Moore, J.S.: Acl2: An industrial strength version of nqthm. In: Proceedings of 11th Annual Conference on Computer Assurance. COMPASS'96. pp. 23–34. IEEE (1996)
7. Keller, R.M.: Formal verification of parallel programs. Communications of the ACM **19**(7), 371–384 (1976)
8. Kroening, D., Paul, W.J., Mueller, S.M.: Proving the correctness of pipelined micro-architectures. In: MBMV. pp. 89–98 (2000)
9. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers 16. pp. 348–370. Springer (2010)
10. 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 **39**(5), 8–16 (2019)
11. Nvidia: Nvidia deep learning accelerator (2018), http://nvdla.org/primer.html
12. Pnueli, A., Shankar, N., Singerman, E.: Fair synchronous transition systems and their liveness proofs. In: Formal Techniques in Real-Time and Fault-Tolerant Systems: 5th International Symposium, FTRTFT'98 Lyngby, Denmark, September 14–18, 1998 Proceedings 5. pp. 198–209. Springer (1998)
13. Rao, N., Bhagavatula, S., Amer, M.R., Ali, K., Lugato, D., Krishnamoorthy, S., Menon, R., Khosrowshahi, A.: Intel nervana: A next-generation neural network processor. In: Proceedings of the 2017 IEEE Hot Chips Symposium on High Performance Chips (HOTCHIPS). pp. 1–28. IEEE, Cupertino, CA, USA (Aug 2017)
14. Sawada, J., Hunt Jr, W.A.: Processor verification with precise exceptions and speculative execution. In: CAV. vol. 98, pp. 135–146 (1998)
15. Vijayaraghavan, M., Chlipala, A., Dave, N.: Modular deductive verification of multiprocessor hardware designs. In: Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II 27. pp. 109–127. Springer (2015)