

Distributed Time-Triggered Caching and Memory Access Optimisation for Neural Network Tensor Accelerators in Multicore Safety-Critical Systems

DISSERTATION
zur Erlangung des Grades eines Doktors
der Ingenieurwissenschaften

vorgelegt von
Aniebiet Micheal Ezekiel, M.Sc.



eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen
Siegen 2025

Betreuer und erster Gutachter
Prof. Dr.-Ing. Roman Obermaisser
Universität Siegen

Zweiter Gutachter
Prof. Dr. Bourdillon Omijeh
University of Port Harcourt, Nigeria

Tag der mündlichen Prüfung
July 14, 2025

Abstract

Neural network accelerators are essential for meeting the computational demands of modern AI applications; however, their use in safety-critical and real-time environments presents significant challenges, primarily due to inefficiencies in memory access and interference from other applications, leading to unpredictable memory access patterns. This dissertation addresses these memory access bottlenecks by proposing a time-triggered architecture that enhances the memory access mechanisms of tensor accelerators. Traditional accelerators, such as the Versatile Tensor Accelerator (VTA), encounter limitations related to memory bandwidth, resource contention, and variable latency, which impair performance in safety-critical, memory-intensive tasks. This work introduces the Time-Triggered Memory Access VTA (TTmaVTA), which applies time-triggered architectures to control and regulate the memory access of the VTA, ensuring predictable and conflict-free memory transactions.

The TTmaVTA framework is further refined with dual memory optimisation techniques, prefetching and caching mechanisms. These enhancements, collectively referred to as OPTTmaVTA, improve memory throughput while significantly reducing memory access latency. Prefetching mechanisms retrieve data during idle memory cycles, minimising delays due to dependency stalls, while deterministic caching optimises frequently accessed memory operations, reducing memory bus accesses. Together, these methods improve the memory performance of neural network accelerators while ensuring timing predictability, particularly in safety-critical contexts.

This dissertation presents hardware experiments and software simulations that validate the effectiveness of TTmaVTA and OPTTmaVTA in improving memory access predictability and memory throughput. Hardware-based experiments using a Conv2D workload on an FPGA demonstrate that TTmaVTA achieves a 2.86% reduction in execution time, primarily due to improved memory scheduling and conflict resolution; however, resource overhead limits TTmaVTA scalability for larger workloads. Software simulations of OPTTmaVTA with ResNet-18 show a 12.68% improvement in memory access time through prefetching and an 8.75% gain through caching. Overall, the OPTTmaVTA architecture achieves improved memory throughput, with a total latency reduction of approximately 19.86% across all memory operations.

In culmination, a scheduling algorithm maps memory access patterns to predefined schedules, ensuring deterministic execution and adherence to real-time constraints. Through a combination of theoretical analysis and practical evaluations, this work makes a substantial contribution to hardware-software co-design for neural network accelerators, particularly suited for applications in safety-critical domains.

Zusammenfassung

Neuronale Netzwerkbeschleuniger sind entscheidend, um den Rechenanforderungen moderner KI-Anwendungen gerecht zu werden. Ihre Nutzung in sicherheitskritischen und Echtzeitumgebungen stellt jedoch erhebliche Herausforderungen dar, insbesondere durch Ineffizienzen beim Speicherzugriff und Interferenzen durch andere Anwendungen, die zu unvorhersehbaren Speicherzugriffsmustern führen. Diese Dissertation adressiert diese Engpässe, indem sie eine zeitgesteuerte Architektur vorgeschlägt, die die Speicherzugriffsmechanismen von Tensor-Beschleunigern verbessert. Traditionelle Beschleuniger, wie der Versatile Tensor Accelerator (VTA), stoßen bei speicherintensiven, sicherheitskritischen Aufgaben auf Einschränkungen in Bezug auf Speicherbandbreite, Ressourcenengpässe und variable Latenzzeiten. In dieser Arbeit wird der Time-Triggered Memory Access VTA (TTmaVTA) vorgestellt, der zeitgesteuerte Architekturen verwendet, um den Speicherzugriff des VTA zu steuern und dabei vorhersehbare, konfliktfreie Speichertransaktionen zu gewährleisten.

Das TTmaVTA-Framework wird durch zwei Speicheroptimierungstechniken, Prefetching und Caching, weiter verbessert. Diese Optimierungen, zusammengefasst als OPTTmaVTA, steigern den Speicherdurchsatz und reduzieren die Speicherzugriffs-Latenzzeiten erheblich. Prefetching-Mechanismen rufen Daten während inaktiver Speicherzyklen ab, um Verzögerungen durch Abhängigkeitsblockaden zu minimieren, während deterministisches Caching häufig genutzte Speicheroperationen optimiert und die Anzahl der Speicherbuszugriffe verringert. Diese Methoden steigern gemeinsam die Speicherleistung neuronaler Netzwerkbeschleuniger und gewährleisten gleichzeitig zeitliche Vorhersehbarkeit, insbesondere in sicherheitskritischen Kontexten.

Die Dissertation präsentiert Hardware-Experimente und Software-Simulationen, die die Effektivität von TTmaVTA und OPTTmaVTA zur Verbesserung der Speicherzugriffsvorhersagbarkeit und des Speicherdurchsatzes bestätigen. Hardware-Experimente mit einem Conv2D-Workload auf einem FPGA zeigen, dass TTmaVTA eine Reduzierung der Ausführungszeit um 2,86% erreicht, hauptsächlich durch optimierte Speicherplanung und Konfliktlösung, jedoch begrenzt der Ressourcenaufwand die Skalierbarkeit für größere Workloads. Software-Simulationen des OPTTmaVTA mit ResNet-18 zeigen eine Verbesserung der Speicherzugriffszeit um 12,68% durch Prefetching und einen Gewinn von 8,75% durch Caching. Insgesamt erreicht die OPTTmaVTA-Architektur eine Steigerung des Speicherdurchsatzes mit einer Gesamtverringern der Latenzzeit um etwa 19,86% über alle Speicheroperationen.

Ein abschließender Scheduling-Algorithmus weist Speicherzugriffsmuster vordefinierten Zeitplänen zu, was eine deterministische Ausführung und Einhaltung von Echtzeit-Anforderungen sicherstellt. Durch eine Kombination von theoretischer Analyse und praktischen Evaluierungen leistet diese Arbeit einen bedeutenden Beitrag zum Hardware-Software-Co-Design für neuronale Netzwerkbeschleuniger, insbesondere für sicherheitskritische Anwendungen.

Acknowledgements

First and foremost, I give all glory to God Almighty for His guidance, wisdom, and strength throughout the course of this work. Without His grace, this dissertation would not have been possible.

I would like to express my deepest gratitude to my supervisor, Prof. Dr. Roman Obermaisser, for his invaluable guidance, continuous support, and encouragement throughout my research. His expertise and insightful feedback have been crucial in shaping this dissertation and my academic journey.

I am also very thankful to the University of Siegen and the Embedded Systems research group for providing a conducive environment and the necessary resources to carry out this research. A special thank you goes to Dr. Daniel Onwuchekwa and Mr. Yosab Bebawy for the stimulating discussions, encouragement, and support both academically and personally during the course of this work.

I would like to extend my heartfelt thanks to Mr. Chigozie Nnah, Dr. Adewale Osineye and Engr. Jacob Usanga, whose support and close motivation inspired me to aim academically higher and pursue excellence in my research.

This research was financially supported by the Federal Ministry of Education and Research (Germany). Contributions to the KIRETT project were also provided by CRS Medical GmbH (Aßlar, Germany), mbeder GmbH (Siegen, Germany), and the associative partners: Kreis Siegen-Wittgenstein, the City of Siegen, the German Red Cross Siegen (DRK), and the Jung-Stilling-Hospital in Siegen.

Further funding was received through the MASON project, carried out within the framework of the Industrial Collective Research Programme (IGF No. 01IF23219N). This project was supported by the Federal Ministry for Economic Affairs and Climate Action (BMWK), based on a resolution of the German Bundestag. Contributions to the MASON project was also provided by the Institut für Unternehmen-skybernetik e. V., RWTH Aachen University.

I am eternally grateful to my parents, Mr. & Mrs. Etok, for their unwavering support, both morally, spiritually, and financially. Their belief in me has been a constant source of strength throughout this journey.

Finally, I express my deepest love and appreciation to my family members for their prayers, especially my wife, Mrs. Barikpoa Progress Ezekiel, and my children, Micah and Michelle, for standing by me and understanding despite my intense absence during this period. Your patience, love, and sacrifices have made this possible, and for that, I am forever grateful.

Contents

1	Introduction	1
1.1	Motivation	4
1.2	Research Objectives and Scope	5
1.3	Problem Definition	6
1.4	Novelty and Contributions	7
1.5	Dissertation Structure	8
1.6	Publications	9
2	Background and Related Work	11
2.1	Artificial Intelligence (AI) Essentials	11
2.1.1	Artificial Neural Networks (ANNs), Deep Neural Networks (DNNs), and Convolutional Neural Networks (CNNs)	11
2.1.2	Design and Training of Neural Networks and Considerations for Safety-Critical Applications	13
2.2	Hardware Platforms for Neural Networks Deployment and Acceleration	15
2.2.1	Graphics Processing Units (GPUs)	17
2.2.2	Edge Artificial Intelligence (Edge AI)	19
2.2.3	Field-Programmable Gate Arrays (FPGAs)	21
2.3	Deep Learning Compilers	25
2.4	Common Deep Learning Hardware and Inference Acceleration Frameworks	29
2.4.1	Apache TVM-VTA (Versatile Tensor Accelerator)	30
2.4.2	Vitis Artificial Intelligence (Vitis AI)	30
2.4.3	Fast, Scalable Quantised Neural Network Inference (FINN)	30
2.4.4	High-Level Synthesis for Machine Learning (HLS4ML)	31
2.4.5	Versatile Tensor Accelerator (VTA)	31
2.4.5.1	Instruction Set Architecture	31
2.4.5.2	Versatile Tensor Accelerator (VTA) Structure	32
2.4.5.3	FPGA Quantisation Processes	34
2.5	Requirements in Safety-Critical Artificial Intelligence (AI) Applications	35
2.5.1	Dependability in Memory Access for Safety-Critical Applications	37
2.5.2	Predictability of memory accesses in safety-critical systems	38
2.6	Memory Access issues in Safety-Critical Applications	40
2.6.1	Hardware Deployment Platforms: Memory Access related Issues for Neural Network Deployment	40
2.6.2	Existing Hardware Frameworks for Inference Acceleration: Unsuitability for Safety-Critical Applications	41
2.6.3	Memory Interference in Neural Network Deployment	42

2.7	Memory Access Management and Optimisation: A Review	43
2.7.1	Researches on Neural Networks towards Reliability in Safety-Critical Applications	44
2.7.2	Researches on Hardware Tensor Accelerators for improving Neural Networks Deployment	45
2.7.3	Researches on Neural Network Memory Interference	46
2.7.4	Researches on Memory Access Optimisations	47
2.8	Leveraging Time-Triggered Approaches for Predictability	49
2.9	Contribution to Memory Access Predictability and Optimisation . . .	50
3	Time-Triggered Caching	53
3.1	Time-Triggered Caching System Model	53
3.2	Time-Triggered Memory Access of Tensor Accelerators	55
3.3	Tracing and Logging of Memory Access Transactions	57
3.3.1	Priming Phase: On-chip Memory Access Tracing of Neural Networks	60
3.3.1.1	Fetch Runtime Synchronised Clock (FRSC)	61
3.3.1.2	Sub-Tracer Modules	62
3.3.1.3	BRAM Cache	64
3.3.2	Offline Phase: Examination and Analysis of Traced Memory Access Data	64
3.4	Memory Access Plot Representation and Analysis	66
3.4.1	Network Flow Graph Representation of Memory Accesses . . .	67
3.4.2	Memory Access Pattern and Latency Analysis	70
3.4.3	Insight into Memory Access Behaviour	74
3.5	Time-Triggered Scheduling	77
3.6	System Application Model: Time-Triggered Memory Access VTA (TTmaVTA)	80
4	Optimisation of Neural Network Memory Accesses	87
4.1	Memory Optimisation Techniques and Strategies	87
4.2	VTA with Optimised Time-Triggered Memory Access (OpTTmaVTA)	89
4.2.1	Idle Memory Phase Utilisation with Data Prefetching in Memory Access	89
4.2.2	Cache-Based Optimisation of Repetitive Memory Access . . .	91
4.2.2.1	Caching Frequently Accessed Data	92
4.2.2.2	Cache Size and Bandwidth Allocation	93
4.2.2.3	Total Time Saved by Caching	93
4.2.3	Reduction in Idle Time	95
4.3	Software Simulation of Versatile Tensor Accelerator (VTA) Operations: Functional Modelling for Optimisation	96
4.3.1	VTA Software Simulator: An Architectural and Functional Review	96
4.3.1.1	Pre-Simulation Phase: Acquisition of VTA Runtime Statistical Data for Simulator Input Generation . . .	97
4.3.1.2	VTA Simulator Component Architecture	99
4.3.2	Extension of the VTA Simulator for Enhanced Memory Handling and Access Optimisation	100

4.4	Optimising Memory Access and Throughput with the Extended VTA Simulator	102
4.4.1	Simulation-based Analysis of VTA Memory Operations	102
4.4.2	Memory Contention Elimination	103
4.4.3	Optimising Memory Utilisation through Idle Time Management Using a Data Prefetching Mechanism	105
4.4.4	Optimising Memory Access with Intermediate Caching of Repetitive Operations	107
4.4.4.1	Caching Strategy for Enhancing Memory Access Efficiency	107
4.4.4.2	Caching Performance Improvement and Time Efficiency Computation	112
5	Experimental Setups, Test Evaluations, and Results	117
5.1	Hardware-Based Experimental Evaluation (TTmaVTA)	117
5.1.1	Memory Access Tracing and Time-Triggered Scheduling	118
5.1.2	Hardware Setup and Testbench Configuration	118
5.1.3	Addressing Memory Contention in VTA Memory Operations .	119
5.1.4	TTmaVTA Performance Analysis	121
5.1.4.1	Execution Time	121
5.1.4.2	Memory Throughput	121
5.1.4.3	Latency Analysis	122
5.1.4.4	Memory Accesses Latency Computation and Comparison between Baseline VTA and TTmaVTA	123
5.1.4.5	Resource Utilisation Analysis	124
5.1.5	Performance Evaluation and Discussion	125
5.2	Software-Based Evaluation of Optimisation Strategies (OPTTmaVTA)	126
5.3	Evaluating Data Prefetching Optimisation Approach	127
5.3.1	Memory Idle Phase Analysis and Prefetching Strategy	127
5.3.1.1	Identification and Characterisation of Idle Phases . .	127
5.3.1.2	Analysis of Prefetchable Operations	128
5.3.2	Computation and Evaluation of the Impact of Prefetching . .	129
5.4	Intermediate Caching of Repetitive Memory Tasks	131
5.4.1	Evaluation of Caching Optimisation	131
5.4.1.1	Optimal Caching Threshold	132
5.4.1.2	Cache Resource Management and Address Allocation	132
5.4.1.3	Load Input Operations: Structure, Cache Access Latencies, and Burst Rates	133
5.4.1.4	Cache Depth Sizing and Cache Address Computation	134
5.4.1.5	Optimisation of Cache Invalidation and Reallocation	135
5.4.2	Performance Evaluation of Cache Optimisation	139
5.4.2.1	Total Time Saved by Caching	139
5.4.2.2	Bus Utilisation Improvement	140
5.5	Evaluating Comprehensive Enhancement of the Incorporated Optimisation Strategies	142
5.5.1	Evaluation of Incorporated Prefetching and Caching Strategies	143

5.5.2	Performance Results	143
5.5.3	Absolute Time Reduction and Performance Improvement Com- putations (OPTTmaVTA)	145
5.6	Limitations and Future Work	145
Conclusion		147
Bibliography		150
List of Acronyms		165
Appendix		167
A	State-Machine of Time-Triggered Memory Access VTA (TTmaVTA)	167
B	Caching Algorithm for Optimisation	168
C	Prefetching Algorithm for Optimisation	169

List of Figures

2.1	Classification of Artificial Intelligence	12
2.2	Convolutional Neural Networks (CNNs) Structure [37].	13
2.3	Convolution Operation Applied to Each Window of the Input Feature Map. [37].	13
2.4	Modern GPUs architecture, a simplified view. Adapted from [11]. . .	18
2.5	Coral Edge TPU architecture [61].	19
2.6	FPGAs architecture, a simplified view	22
2.7	TVM Compiler Stack [77].	28
2.8	TVM IP Structure Framework	29
2.9	VTA Instructions [77].	32
2.10	VTA Data Flow [77].	32
2.11	VTA Structure [77].	33
3.1	System Architecture of Time-Triggered Memory Cache	54
3.2	Time-Triggered Memory Access Architecture	56
3.3	Memory Access Tracer and Logger Unit	59
3.4	Simplified Conventional VTA Design	60
3.5	Custom Logger Intellectual Property (CLIP)	63
3.6	VTA Layer-wise Flow Graph Representation	69
3.7	VTA memory pattern Address Graph Representation	70
3.8	Typical VTA Operation pattern	74
3.9	Memory Access Contention in ResNet-18 Block 1, Layer 1	76
3.10	Schedule Frame	80
3.11	Time-Triggered Memory Access VTA (TTmaVTA) Co-Design	81
3.12	Time-Triggered Memory Access Controller (TTmaC) Block Diagram	82
3.13	Time-Triggered Memory Access Controller (TTmaC)	84
3.14	Schedule Dispatcher State Machine	85
4.1	Versatile Tensor Accelerator (VTA) Simulator	96
4.2	Versatile Tensor Accelerator (VTA) Simulator Extension	100
4.3	A Single Memory Address <i>Cache Pairs</i> Graph showing inter-layer Dependency	107
4.4	Optimised Schedule Frame	115
5.1	Memory Operation Execution for VTA Layer-1: Stage-1	119
5.2	Optimised Memory Access Operation without Contention	120
5.3	Performance Comparison of Memory Access Time, Memory Throughput, and Latency between Conventional VTA and TTmaVTA.	122
5.4	Latency dispersion curve comparison between VTA and TTmaVTA.	124

5.5	Resource utilisation comparison between VTA and TTmaVTA.	125
5.6	Memory Latency Extracts of the First Layer.	128
5.7	Prefetching Performance Evaluation.	129
5.8	Memory Latency Comparison across Computational Layers of TTmaVTA and OPTTmaVTA with Prefetching.	130
5.9	Latency of a single write with multiple read accesses to the same memory address.	132
5.10	Graph of multiple writes and reads to address 0x06090000.	136
5.11	Cache Size Progression: No Invalidation vs. Invalidation Scenarios. .	139
5.12	Caching Performance Evaluation.	141
5.13	Memory Latency Comparison across Computational Layers of TTmaVTA and OPTTmaVTA with Caching.	141
5.14	ResNet-18 Memory Latency Trend across Computational Layers for TTmaVTA and OPTTmaVTA.	144
15	TTmaVTA Task Coding State-machine	167

List of Tables

1.1	List of Publications	10
2.1	Comparative analysis of hardware platforms.	24
2.2	Comparison of Deep Learning Compilers	26
5.1	Execution Time Comparison between Conventional VTA and TTmaVTA	121
5.2	Memory Throughput Comparison between Conventional VTA and TTmaVTA	122
5.3	Latency Comparison between Conventional VTA and TTmaVTA . .	122
5.4	Prefetchable and Non-Prefetchable LOAD WGT Operations across Layer Groups	129
5.5	LOAD INP Operational Statistics	133
5.6	STORE Operations <i>Cache Pair</i> Statistical Structure.	134
5.7	Cache operations showing Cache Invalidations (Inv_IDs)	137

Chapter 1

Introduction

The contemporary landscape has seen a notable shift towards Artificial Intelligence (AI), marked by a significant surge in the adoption and utilisation of neural networks. These networks have effectively tackled various challenges, enhancing processes across diverse domains. However, deploying such systems in specific contexts necessitates strict consideration of safety and reliability, particularly in Critical Autonomous AI-based Systems (CAIS) and safety-critical applications. Substantial research efforts are directed towards enhancing the safety and reliability of AI systems, focusing on aligning AI-embedded applications with standards in highly reactive and concurrent sectors such as aerospace, medical, automotive, and nuclear power [1, 2].

The increasing reliance on neural networks (NNs) for critical applications necessitates a comprehensive understanding of their deployment, particularly using hardware accelerators in domains where safety is paramount. Neural networks-based applications have shown significant potential across various fields and are gaining traction in safety-critical domains such as healthcare [3], automotive [4], avionics [5], space [6], railways [7], and industry [8]. The research community has recognised the importance of dependable and predictable systems for highly accurate and reliable applications. These demands are especially pronounced in safety-critical domains, where failure can have significant consequences, necessitating robust deployment strategies for neural networks.

Deploying neural networks, such as Convolutional Neural Networks (CNNs), on resource-constrained devices like Field-Programmable Gate Arrays (FPGAs) has become increasingly prevalent due to their advantages in reducing communication latency and improving safety and security by handling sensitive information locally [9, 10]. However, existing tensor accelerators are less suitable for safety-critical applications [11]. Designing appropriate toolflows for neural network deployment requires a careful trade-off between performance, power consumption, and programmability.

Deploying deep learning techniques in safety-critical applications is challenging due to stringent real-time and reliability requirements. Traditional hardware platforms for neural network acceleration, such as FPGAs, offer promising solutions due to their reconfigurability and efficiency. However, the deployment process for these models on hardware accelerators is still maturing and requires significant development to meet the necessary standards. Safety-critical systems demand techniques to manage runtime unpredictability, often requiring additional functional blocks for timing or energy management [12].

Deploying neural networks in FPGAs for safety-critical applications has gar-

nered noticeable interest. However, applications in this domain require safety and predictability of runtime behaviour [13]. AI-enabled embedded systems are prone to uncertainty, producing outputs with uncertain correctness, making them non-deterministic and raising concerns for safety and reliability [14].

AI-enabled embedded systems are often prone to significant uncertainty, as their components produce functional outputs with uncertain outcomes regarding their correctness, making them non-deterministic [12] and posing a deep concern for safety and reliability. These issues related to safety and predictability limit AI applications in the safety-critical domain [14], as these applications require mechanisms to avoid catastrophic outcomes in the event of failures [15]. One of the primary challenges in deploying neural networks in safety-critical applications is ensuring their predictability and reliability. Successful implementation in environments such as real-time monitoring or diagnostic imaging provides substantial validation to the field of neural network applications. Such validation can enhance trust and broader adoption of these technologies in other critical domains.

Common challenges in deploying neural networks using traditional hardware platforms in safety-critical applications include:

- The intricate design cycle required to translate a problem into a deployable neural network model on hardware accelerators. Machine and deep learning research has primarily focused on developing and testing new models for non-critical applications. As a result, hardware designs suited for safety-critical domains often need to catch up. The challenge lies in designing and deploying these models efficiently, accessibly, and reliably [16].
- The lack of environment adaptability and reconfigurability. Current static deep learning systems offer substantial opportunities for data processing in critical applications, but their inflexibility can pose significant risks and operational constraints. Therefore, it is crucial to adopt adaptive and reconfigurable deep learning strategies tailored to the specific needs of these applications.
- Issues related to memory access also affect AI implementation using these hardware platforms. Neural networks, especially deep learning models, require significant memory bandwidth and efficient memory access patterns to function correctly [17] [18]. In safety-critical applications, unpredictable memory access can lead to non-deterministic behaviour, which is unacceptable.

Moreover, a significant gap exists between the Functional Safety (FUSA) requirements in the safety-critical domain and the inherent nature of deep learning hardware accelerators [19]. Existing accelerators, such as the Versatile Tensor Accelerator (VTA), lack inherent capabilities to deliver predictable performance, rendering them less suitable for safety-critical real-time applications where human lives are at risk.

Neural networks introduce uncertainties from memory-level interference, including memory conflicts, data movement, task dependencies, algorithmic complexity, and dynamic workloads. Meeting stringent timing requirements while accommodating the computational demands of neural networks poses a challenge in ensuring predictable and reliable system behaviour. A central concern pertains to the development process of deep learning libraries, which influences high-level safety concepts such as explainability and traceability [20].

Unpredictable memory access behaviour during neural network deployment and acceleration further complicates their suitability for safety requirements and real-time constraints [21]. Inefficient resource utilisation exacerbates these challenges, possibly resulting in wasted computational capacity and execution delays in resource-constrained systems. These issues have the potential for catastrophic failures, increasing the risk of safety and reliability breaches and impeding timely execution.

Significant memory resources are required to store AI parameters such as weights, inputs, and operations for state-of-the-art neural networks. However, most resource-constrained embedded systems rarely meet these requirements [22]. As the complexity of neural network architectures increases, memory-related issues compound, making memory access optimisation crucial for achieving high performance. Memory-related issues observed in existing accelerators induce resource contention [23], variable latency [24], and unpredictable behaviour [25], leading to underutilisation of memory bandwidth [26], and unjust access during concurrent Dynamic Random Access Memory (DRAM) access attempts. Consequently, safety-critical embedded systems face significant risks due to unpredictable memory accesses, potentially resulting in missed deadlines and system failures. Memory access delays in existing tensor accelerators, caused by unfulfilled memory operations dependencies stalling the memory bus, compromise memory resource utilisation, memory access efficiency, system predictability, and reliability.

These limitations necessitate the use of adaptive strategies such as time-triggered methods [27], which have shown promising results in extending the common tensor accelerators to overcome safety limitations. Several safety-critical applications have established safety arguments by leveraging time-triggered strategies [27]. Extending existing Tensor Accelerators like the VTA with time-triggered components is promising in overcoming safety limitations. This work proposes a time-triggered approach to attain temporal predictability in VTA while guaranteeing safety and mitigating random faults in embedded AI platforms.

The time-triggered architecture emerges as a promising solution, offering predictability and determinism [28]. This architecture provides tight control over system responses [29], ensuring that critical tasks, such as neural network inferences, consistently execute within predefined time windows. It facilitates resource management with low runtime overheads [30], allowing for controlled and predictable resource allocation to neural networks, thereby ensuring timely execution. A pivotal element of a time-triggered system is the schedule, which is essential for ensuring the timely execution of neural networks in safety-critical applications. Consequently, this dissertation's first contribution involves generating a robust schedule for integration into a time-triggered architecture, maximising the utilisation of hardware resources and ensuring system predictability. Additionally, the efficiency of memory operations is enhanced by prefetching data for non-dependent memory operations during idle windows, leading to improved memory access performance.

Adapting the proposed memory access technique with a time-triggered schedule offers a more predictable and energy-efficient computing system. Thus, careful system design, thorough testing, and validation processes are essential. The dissertation commences with a comprehensive examination of existing accelerators, mainly focusing on their memory operations to discern patterns and identify potential gaps. The proposed time-triggered memory access algorithm aims to address these issues by ensuring predictable memory access patterns and enhancing the reliability and

performance of neural network deployment in these applications.

Two primary methodologies have been employed for acquiring, tracing, and logging memory access information (intrinsic) from hardware accelerators: hardware-based and software-based memory acquisition. Hardware-based methods entail integrating memory blocks such as Block RAM and data caches to capture and preserve information about the accelerator’s memory accesses [2]. Conversely, the software method involves introducing a flag into the software compiler stack to dump memory access information onto the hardware memory. This dissertation explores both approaches: leveraging runtime information trace analysis and adopting a simulation-based method with scheduling capabilities to alleviate queuing constraints while prioritising memory requests.

This work aims to develop tools that facilitate the deployment of deep neural networks, such as CNNs or ANNs, on existing tensor accelerators like the VTA deployed on hardware platforms such as FPGAs. Additionally, it explores how deep learning systems conscious of memory access can function in safety-critical applications. These systems will be based on existing terrestrial deep learning models, with modifications to meet stringent requirements and constraints of safety-critical environments. The goal is to create a workflow that implements these models efficiently and reliably, ensuring their time predictability and safety for critical applications. Incorporating a time-triggered memory access algorithm with memory access optimisation strategies on existing tensor accelerators will further enhance the predictability and robustness of these deployments, addressing the critical issue of memory access in AI implementations within this domain.

1.1 Motivation

Imagine the catastrophic impact of a failure in healthcare systems, where a misdiagnosis due to unpredictable neural network behaviour could result in incorrect treatments or missed diagnoses, potentially leading to severe health consequences or loss of life. Similarly, consider the implications in autonomous driving systems, where delays or errors in decision-making due to unpredictable neural network performance could result in accidents, endangering lives. The potential for having a predictable neural network accelerator with deterministic memory access could significantly mitigate these risks. Such a system would ensure critical tasks are executed within defined time constraints, enhancing safety and effectiveness in these high-stakes environments.

To meet the stringent requirements of safety-critical applications, it is essential to achieve high predictability and reliability in neural network deployments using hardware accelerators. Safety-critical domains, such as medical diagnostics, autonomous driving, aerospace, and industrial control, require systems that continuously monitor their environment and make real-time decisions based on neural network outputs. Traditional methods relying on centralised processing and human intervention to manage large datasets are no longer feasible. These methods are often time-consuming and resource-intensive, making them less suitable for applications where decisions must be made within narrowly defined time windows.

Integrating deep learning breakthroughs with predictable tensor accelerators represents a crucial advancement. Time-triggered architectures, for instance, provide deterministic execution and temporal predictability by scheduling tasks within pre-

defined time windows. This approach is particularly relevant for safety-critical applications, where the reliability and timing of neural network operations are paramount. This dissertation addresses these key challenges in deploying neural networks effectively in these demanding environments by combining advances in deep learning with hardware solutions that ensure predictable performance.

This research is pivotal as it fills a critical gap in ensuring that neural network deployments meet the rigorous standards required for safety-critical applications. The research aims to develop and refine methods to enhance predictability and reliability in neural network accelerators, making them suitable for applications where failure is not an option. This work will not only facilitate the broader adoption of neural networks in safety-critical domains but also drive advancements in related fields where reliability and predictability are essential.

Furthermore, the findings and innovations from this research will have significant impacts beyond safety-critical applications. Enhanced memory access predictability and optimisation techniques have shown great potential in improving neural network accelerators' performance in various domains, including consumer electronics, robotics, and edge computing. By contributing to developing more reliable and predictable hardware accelerators, this research supports the advancement of artificial intelligence technologies across a wide range of applications, ultimately revolutionising the landscape of safety-critical systems and beyond.

1.2 Research Objectives and Scope

Recent advancements in deep learning have led to significant improvements in performance metrics, with a predominant focus on accuracy. This trend is largely driven by research centred around established benchmarks such as ImageNet and Pascal Visual Object Classes, which emphasise offline performance. However, there is an emerging trend in the literature to adapt this research to embedded systems and address the novel scientific challenges that arise in these contexts. To advance neural network adaptations in the field of safety-critical applications, it is essential to engage in collaborative efforts to identify the most pressing needs for new technology and the challenges associated with its adoption.

One critical area that remains underexplored is the development of suitable tensor accelerators to deploy deep learning models within safety-critical environments. These systems must autonomously manage data reliability, dependability, and timing guarantees with predictable execution times of memory accesses to ensure robust performance. This research focuses on predictably deploying neural network applications using modern neural networks (ANNs, CNNs and others) on existing tensor accelerators such as the VTA deployed on hardware platforms such as FPGAs with limited resources for safety-critical applications.

Specifically, this dissertation aims to:

- Trace and investigate the intrinsics of an existing tensor accelerator's memory accesses of deep learning systems at runtime deployment on hardware (FPGAs).
- Study the traced memory accesses to develop a time-triggered memory access controller with an algorithm that enhances the predictability of neural network deployment on existing tensor accelerators.

- Create a co-design workflow of an existing accelerator that facilitates the efficient and reliable deployment of neural networks on FPGAs, addressing the critical issues of memory access patterns and system predictability.
- Incorporate memory access optimisation techniques such as prefetching data and caching of repetitive access data to improve memory bandwidth and reduce memory throughput while improving the inference time.
- Evaluate the system’s operation within the power and resource constraints typical of safety-critical applications to deliver reliable performance even under variable conditions.

By addressing these objectives, this research will contribute to advancing neural network deployment in safety-critical applications, ensuring that these systems can meet the high reliability and predictability standards required in such environments.

This dissertation does not cover or consider in detail some aspects of safety-critical requirements such as fault detection and fault tolerance, core energy efficiency evaluations, and some aspects of safety and security. Furthermore, some solutions reported, such as caching repetitive memory access data, were limited to testing on a software simulation test bench and are subject to further complete hardware deployment evaluations.

1.3 Problem Definition

Deploying neural networks in safety-critical applications presents several challenges, particularly in ensuring the predictability and reliability of the system. Traditional approaches to neural network deployment often focus on maximising performance and accuracy, with less emphasis on the specific requirements of safety-critical environments. These environments demand stringent guarantees on timing and behaviour, as any deviation can lead to catastrophic outcomes [31].

Deep learning accelerators have been widely employed to deploy neural networks across various hardware platforms. However, existing solutions that aim to deploy Deep Neural Networks (DNNs) on energy-efficient and resource-constrained edge devices, such as FPGAs, often prove inefficient [32]. These inefficiencies arise from various factors, creating significant challenges that act as barriers across multiple constraint levels within the hardware/software stack, thereby impeding the efficient execution of DNNs on these platforms [33].

Among the numerous challenges, the primary issue addressed in this research is the need for predictability in memory accesses and its unreliability in deploying neural networks using existing hardware Tensor accelerators on FPGAs, which is crucial for safety-critical applications. This problem encompasses several key challenges:

1. **Timing Unpredictability:** Ensuring that neural network operations are executed within predefined time windows is crucial in safety-critical systems. Traditional deployment approaches may not provide the necessary guarantees on timing, leading to unpredictable behaviour and potential system failures.
2. **Resource Constraints:** Safety-critical applications often operate in resource-constrained environments with limited power, memory, and computational ca-

capacity. Efficiently mapping neural networks to these constrained resources while maintaining performance poses a significant challenge.

3. **Memory Access Patterns:** Memory interference in neural network accelerators is a persistent obstacle, particularly in safety-critical applications where deterministic behaviour is paramount. Uncontrolled memory interference can cause contention and delays in data retrieval, leading to unpredictable system behaviour. Furthermore, neural networks require significant memory bandwidth for storing and accessing parameters, inputs, and intermediate data. Unpredictable memory access patterns can lead to non-deterministic behaviour, which is unacceptable in safety-critical applications. This unpredictability can compromise the timing guarantees essential for maintaining the safety and reliability of these systems.
4. **Adaptability and Reconfigurability:** Safety-critical systems may need to adapt to varying conditions and requirements, necessitating reconfigurable and adaptive neural network implementations. Traditional static approaches to neural network deployment may not provide the necessary flexibility.

To address these challenges, this research proposes a time-triggered approach to neural network deployment on FPGAs using an existing tensor accelerator, ensuring temporal predictability and reliability. This approach involves designing a hardware architecture with a memory controller that supports time-triggered execution, managing data transfers and computations based on a global clock. Additionally, the study explores techniques for efficient mapping neural networks to resource-constrained FPGAs, optimising memory access patterns by reordering memory access executions to improve memory bandwidth and throughput while reducing inference speed.

1.4 Novelty and Contributions

This dissertation advances the deployment of neural networks in safety-critical applications through the following key contributions:

1. Insights into Neural Network Predictability and Reliability

- The dissertation develops and explores memory tracers for memory access data acquisition on existing tensor accelerators to compute memory access patterns, instruction groupings, dependencies, and timing information.
- The traced information is analysed using network flow graphs, charts, and other statistical data representations to gain insights into potential system modifications. This analysis ensures neural network predictability and reliability in safety-critical applications, guiding necessary adjustments to the system architecture and implementation.
- A system is proposed that uses prior knowledge of the accelerator memory access to autonomously manage data access reliably using preconfigured predictable execution times, ensuring robust performance under variable conditions typical of safety-critical domains.

2. Efficient Workflow for Neural Networks Deployment on Resource-Constrained FPGAs

- The workflow developed facilitates efficient and reliable deployment of neural networks on FPGAs.
- The workflow addresses memory access patterns and system predictability issues, ensuring effective operation within the power and resource constraints of safety-critical applications.

3. Time-Triggered Memory Access Control for Neural Network Deployment on FPGAs

- An integrated time-triggered memory access system is developed using a time-triggered algorithm. This system is based on the traced memory access information from the conventional VTA, allowing for the generation of a suitable schedule that provides predictability and timing guarantees.
- This algorithm ensures operations are executed within predefined time windows, enhancing timing predictability and system reliability.

4. Optimisation Techniques for Memory Access in FPGA-Based Neural Networks

- Optimisation techniques, including data prefetching during memory idle windows and caching of repetitive memory access data, are incorporated to optimise memory access patterns. These enhancements improve the efficiency of memory data transfers and computations, contributing to more predictable and reliable system performance.
- These optimisation techniques ensure effective neural network deployment on resource-constrained hardware, increasing speed and improving memory throughput while maintaining high performance.

This dissertation emphasises the importance of time-triggered strategies and resource-efficient designs in addressing the limitations of current neural network deployment approaches in these environments. These contributions collectively address critical challenges in neural network deployment for safety-critical applications, enhancing their reliability, predictability, and efficiency.

1.5 Dissertation Structure

This dissertation is structured as follows:

Chapter 1 introduces the research topic, outlining the motivation and significance of the study, presenting the objectives and research questions, and providing an overview of the dissertation structure.

Chapter 2 presents a comprehensive background and literature review covering neural networks in safety-critical applications, hardware-based deployment for deep learning, especially on FPGAs, challenges of deploying neural networks in resource-constrained environments, and current methodologies for ensuring predictability and reliability in neural network operations. This chapter also provides background and related work in memory access on FPGA-based tensor accelerators. It includes a

comprehensive review of existing tensor accelerators, particularly the VTA and its various modules, and related research in time-triggered architectures.

Chapter 3 details the time-triggered caching, its system model and methodology, describing the proposed approaches for addressing identified challenges. It outlines the co-design of VTA with a time-triggered controller and uses a custom hardware module to log data and analyse memory access information. The chapter further introduces the detailed system architecture of the Time-Triggered Memory Access Versatile Tensor Accelerator (TTmaVTA), including the time-triggered memory access algorithm and mechanism, detailing its design, implementation, and integration with the conventional VTA. It describes the design considerations and implementation details of VTA, covering both hardware and software aspects, and explains the TTmaVTA architecture, the time-triggered Memory Access Controller (TTmaC) design, and the structure of the schedule dispatcher. This chapter also elaborates on the schedule formulation by integrating parametric measurements (memory access time, addresses, and dependencies) into the TTmaC.

Chapter 4 presents a detailed exploration of memory optimisation strategies, focusing on techniques such as prefetching during memory idle phases and caching frequently accessed data through repetitive memory operations. These strategies are integrated into the extended VTA Simulator, already augmented with enhanced logging and post-processing capabilities to support comprehensive analysis. The chapter demonstrates how the extended VTA Simulator effectively captures memory access patterns and operational data, facilitating the evaluation of performance metrics such as latencies, resource utilisation, and scheduling efficiency. By incorporating post-processing modules, the simulator generates detailed execution timelines and schedules that are utilised to assess performance in a simulation environment, which is suitable for evaluating OPTTmaVTA, a memory-optimised variant of the TTmaVTA. Additionally, algorithms for mitigating memory contention and optimising memory operations are introduced, further supporting the development of efficient computational frameworks for time-triggered memory access evaluation.

Chapter 5 then explains the experimental setup and test and evaluation metrics. This chapter provides a comprehensive experimental evaluation and results, analysing and evaluating the TTmaVTA approach on memory access with some overhead runtime considerations and discussing the results of the OPTTmaVTA as simulated on a deep learning testbench. It includes the performance evaluation of the time-triggered tensor accelerator, benchmarking methodology, performance comparisons, and applications demonstrating VTA’s effectiveness for various neural network workloads and algorithmic variations. It further analyses the impact of this dissertation on the predictability and reliability of neural network memory operations on FPGAs.

Finally, the dissertation concludes with a discussion of the findings, implications, limitations, and potential improvements of the VTA architecture, summarising key insights and contributions, and suggesting directions for future research.

1.6 Publications

The work described in this dissertation has been published in the following peer-reviewed journals and conference proceedings as author or co-author:

Table 1.1: List of Publications

No	Publication	Status
1	Ezekiel, A. M., Bebawy, Y., Obermaisser, R., "Enhancing Neural Network Predictability through Simulation-Based Analysis of Memory Accesses for Time-Triggered Tensor Acceleration," 2024 10th International Conference on Control, Decision and Information Technologies (CoDIT), Vallette, Malta, 2024, pp. 2839-2844.	Published
2	Ezekiel, A. M., Onwuchekwa, D., Obermaisser, R., "Optimisation of the Versatile Tensor Accelerator (VTA) Load Module in a Time-Triggered Memory Access," 2023 26th Euromicro Conference on Digital System Design (DSD), 2023, pp. 146-152.	Published
3	Ezekiel, A. M., Obermaisser, R., "Time-Optimised Detection of Cardiovascular Complications with Artificial Intelligence in Rescue Operations using FPGA-based Wearable," 2023 5th International Congress on Human-Computer Interaction, Optimisation and Robotic Applications (HORA), 2023, pp. 1-8.	Published
4	Bebawy, Y., Ezekiel, A. M., and Obermaisser R., "Super-scalar Time-Triggered Versatile-Tensor Accelerator," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 44, no. 7, pp. 2503-2515, July 2025, doi: 10.1109/TCAD.2025.3528355.	Published
5	Bebawy, Y., Schmidt, M. -C., Ahmadian, H., Ezekiel, A. M., Obermaisser, R., "Time-Triggered Inference on FPGAs," 2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFa), Padova, Italy, 2024, pp. 1-7.	Published
6	Ahammed, A. S., Ezekiel, A. M., Obermaisser, R., "Time-Efficient Identification Procedure for Neurological Complications of Rescue Patients in an Emergency Scenario Using Hardware-Accelerated Artificial Intelligence Models," Algorithms, vol. 16, no. 5, article 258, 2023.	Published
7	Ahammed, A. S., Ezekiel, A. M., Obermaisser, R., "A Novel Analysis of Performance and Inference Time of Machine Learning Models to Detect Cardiovascular Emergency Situations of Rescue Patients," 2022 International Conference on Artificial Intelligence of Things (ICAIoT), 2022, pp. 1-6.	Published
8	Zenkert, J., Weber, C., Nadeem, M., Bender, L., Fathi, M., Ahammed, A. S., Ezekiel, A. M., Obermaisser, R., Bradford, M., "KIRETT - A wearable device to support rescue operations using artificial intelligence to improve first aid," 2022 IEEE International Smart Cities Conference (ISC2), 2022, pp. 1-4.	Published

Chapter 2

Background and Related Work

This chapter provides the challenges and advancements in memory access management for neural networks deployed in safety-critical applications. It reviews the limitations of existing tensor accelerators in addressing memory throughput and latency issues that affect their reliability. Research on memory controllers that adapt to varying access patterns is explored, emphasising their potential to improve predictability in memory operations under resource constraints. The chapter outlines a set of proposed techniques for memory access optimisation, including time-triggered memory access, time-based scheduling of memory operations, predictive memory management through prefetching, and deterministic data caching. These strategies are presented as crucial for enhancing the performance and reliability of neural network accelerators in safety-critical environments.

2.1 Artificial Intelligence (AI) Essentials

Artificial Intelligence (AI) is broadly defined as any program capable of sensing, reasoning, acting, and adapting to various conditions [34]. It encompasses any approach or technique that enables computers to mimic human intelligence, using logic, if-then rules, or decision trees. As shown in Figure 2.1, AI includes machine learning as a subset, consisting of algorithms that use statistical techniques to improve performance, such as output accuracy, as they process additional data over time. Deep learning is a further subset of AI algorithms, composed of multilayer neural networks that learn from vast amounts of data. AI, machine learning, and deep learning span multiple categories, with neural networks falling into three primary types: Convolutional Neural Networks (CNNs), Artificial Neural Networks (ANNs), and Deep Neural Networks (DNNs). Each type of neural network serves different purposes: CNNs are typically used for image recognition and processing, ANNs for general-purpose machine learning tasks, and DNNs for more complex and deep-layered learning tasks.

2.1.1 Artificial Neural Networks (ANNs), Deep Neural Networks (DNNs), and Convolutional Neural Networks (CNNs)

A neural network comprises interconnected neurons that perform non-linear processing in response to input signals [35]. It has a structure consisting of layers,

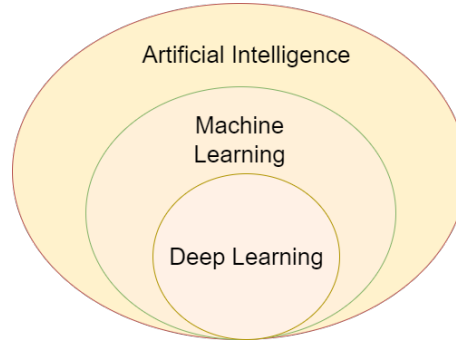


Figure 2.1: Classification of Artificial Intelligence

with each layer containing nodes known as neurons. These neurons receive signals from the neurons in the previous layer, process them, and transmit the output to the neurons in the next layer. Each connection between neurons has an associated weight, which adjusts as the network learns during the training phase.

Artificial Neural Networks (ANNs) are a form of artificial intelligence that have gained renewed research interest over the past decade. While traditionally utilised for various engineering problems, ANNs have recently been applied to a broad range of challenges, especially in the medical domain, with significant impacts in radiology, urology, laboratory medicine, and cardiology.

An Artificial Neural Network (ANN) is a distributed network of computing elements modelled after biological neural systems and can be implemented as a computer software program. These networks are adept at identifying relationships in input data that are not easily apparent with conventional analytic techniques. The functioning of an ANN is built on learning and experience from previous input data. Based on this prior knowledge, the network can predict relationships found in newly presented data sets. For instance, in cardiology, ANNs have been successfully applied to diagnose coronary artery disease and myocardial infarction, interpret electrocardiographic data, detect arrhythmias, and analyse images in cardiac radiography and sonography.

ANNs are universal function approximators capable of solving tasks such as classification, detection, and regression. They utilize backpropagation training [36] and sufficient network complexity to perform effectively. Today, most ANNs used in pattern recognition are composed of several layers, known as deep neural networks (DNNs).

DNNs encompass several classes, with Convolutional Neural Networks (CNNs) being a specific and widely used class, particularly known for their efficiency in tasks such as object detection, image processing, and pattern recognition. CNNs are designed to automatically and adaptively learn spatial hierarchies of features from input images, making them highly effective for visual recognition tasks.

The fundamental components of CNNs include convolutional layers, pooling layers, and fully connected layers. Convolutional layers apply a series of filters to the input image, capturing spatial features and patterns. Pooling layers reduce the dimensionality of the feature maps, making computation more manageable while retaining essential information. Fully connected layers, typically found towards the end of the network, perform the final classification based on the features extracted by the convolutional and pooling layers. Most layers of a CNN perform convolution

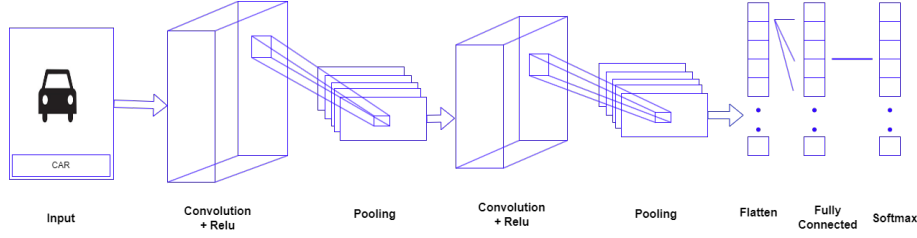


Figure 2.2: Convolutional Neural Networks (CNNs) Structure [37].

as seen in Figure 2.2, applying a filter to an image to extract features (feature maps) used for detecting and classifying objects.

Computationally, a filter is implemented as a kernel, whose values, determined during the training phase, dictate the nature of the layer’s extracted information. As shown in Figure 2.3, the filter is typically convolved with a window of 3×3 elements (floating-point numbers), though the size may vary based on design choices.

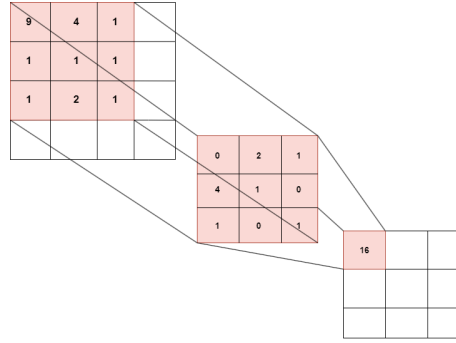


Figure 2.3: Convolution Operation Applied to Each Window of the Input Feature Map. [37].

The choice of convolution kernels is not directly made by the programmer but by the network itself. This involves combining several feature maps to classify the image correctly, resulting in processed images that are not easily recognisable by humans. During the extraction of abstract information from images, high resolution is not necessary, allowing for data reduction after convolution. Convolutional layers are interleaved with pooling layers to filter data propagated to downstream layers, avoiding unnecessary computation. Common pooling methods include average pooling and max pooling, with average pooling propagating the average value of elements in the window and max pooling propagating the element with the highest value. The methods to design and train neural networks, irrespective of the types, become paramount for optimising performance and ensuring accurate, efficient processing in various applications.

2.1.2 Design and Training of Neural Networks and Considerations for Safety-Critical Applications

A key aspect of designing neural networks is identifying the number and types of layers and properly interconnecting them for specific tasks. This is achieved through a training phase, which is computationally expensive and involves tuning network parameters to fit the dataset. Training the network involves forcing it to learn

by processing hundreds of thousands of labelled datasets to identify specific image classes. During training, each training input-output pair is adjusted to ensure the network output is as close as possible to the ground truth, with the loss function recording the distance between true and predicted values. The training of neural networks, particularly deep learning models like CNNs, involves a significant amount of data and computational resources. The process of training adjusts the weights of the connections based on the errors between the predicted and actual outputs, using algorithms like backpropagation [38].

Training is performed through backpropagation of the output error from the last layer back to the input layer. This process involves forwarding a batch of data through the network, computing the loss function gradients, and comparing the output with ground-truth labels. An optimiser, such as stochastic gradient descent [39], minimises the loss function by updating the weights. Training consists of repetitive forward and backward iterations for each batch of training data over multiple epochs (complete passes over the training set). Ensuring good generalisation capabilities of the model requires a highly heterogeneous training dataset and data augmentation strategies to create a more varied training set. This practise helps fill gaps and prevent significant performance drops when the neural network is deployed outside its trained scope.

During the training phase, design choices and training methods can significantly impact the overall performance and suitability of the model for memory-efficient applications. One area of concern is memory-related issues during training, which can affect convergence time, especially in small models, and consequently impact model accuracy. However, it is interesting to note that transient memory issues during neural network training are unlikely to affect the final model’s accuracy, as temporary memory faults during training are mitigated by subsequent training iterations. Proper network design encodes the knowledge required for the task, demonstrating the architecture’s expressivity and trainability. Effective training combines suitable techniques to ensure the neural architecture can solve the task with good performance while tuning all network parameters. A well-designed model and training process have the potential to make the neural network intrinsically more reliable against transient memory issues.

Each neural network design includes a distinct set and arrangement of layers. Convolutional layers, for example, utilise specific hyperparameters such as kernel size (the dimensions of the convolution kernel), stride (the movement increment over the image), and padding (the number of zeros added to the input image borders for edge convolution). Each kernel operates independently, producing a unique feature map, with the number of output feature maps corresponding to the number of filters [40].

In addition to convolutions, which are computationally intensive and sensitive to memory limitations, activation functions are crucial for establishing a nonlinear input-output relationship in DNNs. Rectified linear units (ReLU) are commonly used for this purpose, defined as $\text{ReLU}(x) = \max(0, x)$, where x represents the input tensor. This function facilitates gradient flow, which is essential for training [41]. Variants of ReLU, such as the scaled exponential linear unit (SELU) [42], the Gaussian error linear unit (GELU) [43], and ReLU6 [44], have also been developed. Normalisation layers help stabilise the training of neural architectures by smoothing the optimisation landscape [45] and preventing issues such as weight and gradient

explosion. The most widely used normalisation layer is BatchNorm, which learns an approximation of the first and second statistical moments of each feature map during training to normalise the input tensors. After normalisation, an affine transformation is typically applied, involving an additive bias (β) and a scale parameter (γ). The normalisation operation is defined as:

$$\text{BatchNorm}(x) = \left(\frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} \right) \times \gamma + \beta \quad [46] \quad (2.1)$$

where $\mathbb{E}[x]$ is the expected value of the input tensor x , $\text{Var}[x]$ is its variance, and ϵ is a term added to improve stability.

The output of a neural network is a vector of tensors representing the probabilities of eligible objects. Objects identified with sufficiently high probabilities are classified, allowing the network to detect multiple objects with varying probabilities. To select objects with a high likelihood of actual detection, a probability threshold filter is applied. Setting this threshold is crucial: a high threshold may cause the system to miss existing objects, increasing the risk of collisions, while a low threshold might result in numerous false positives, causing unnecessary stops. In object detection DNNs, a tensor also includes the coordinates of a bounding box, which describes the detected object.

Neural networks training phase is typically computationally intensive, requiring powerful Graphics Processing Units (GPUs) or specialised hardware platforms for acceleration. Due to their high computational demands, neural networks, particularly CNNs, are efficiently mapped to parallel processors. Consequently, they benefit significantly from the use of GPUs or dedicated hardware platforms for computation during both training and inference processes. Once trained, neural networks can be deployed to perform inference on new data using hardware platforms such as GPUs, Field-Programmable Gate Arrays (FPGAs), or Edge Artificial Intelligence (EdgeAI).

The deployment phase, especially on resource-constrained devices like FPGAs, presents unique challenges. These include ensuring that the network runs efficiently within the limited resources available while maintaining the required performance levels. This phase also requires considerations of power consumption, real-time processing capabilities, and system reliability, particularly in safety-critical applications. Maintaining a balance between performance and resource constraints is crucial to ensure effective deployment on these platforms.

2.2 Hardware Platforms for Neural Networks Deployment and Acceleration

Neural networks, including Artificial Neural Networks (ANNs) and Convolutional Neural Networks (CNNs), require substantial computational power for their inference processes. This demand arises from the need to convolve defined filters with each window of every convolutional layer to extract features during training. Safety-critical applications, such as those in the medical domain, necessitate parallel and complex hardware devices to ensure real-time execution of at least 40 frames per second [11].

Research in neural networks and their applications has advanced significantly due to their ability to model complex systems with large inputs both easily and accurately [47]. Neural networks offer several advantages, including high processing speeds achieved through massive parallel implementation [48], as well as valuable properties such as adaptability, self-learning capabilities, nonlinearity, and smooth input-to-output mapping [49]. These networks also enhance vital data analysis elements, including accuracy, performance, convergence, fault tolerance, latency, and scalability [50, 51]. The combination of these advanced capabilities and significant computational demands highlights the importance of continued research and development in neural network technology to meet the needs of diverse and critical applications.

The design of dedicated hardware for neural network execution is challenging due to the high complexity and performance requirements. Commercial Off-The-Shelf (COTS) devices are frequently employed to deploy neural networks at optimal speeds [52]. However, these devices are not specifically tailored for safety-critical applications, requiring significant technological adaptation, integration, and comprehensive evaluation to ensure their suitability. Furthermore, evaluating the reliability of modern COTS devices for neural networks is complex due to the sparse and limited information available about their technology, implementation, and architecture, often necessitating reverse engineering [11].

The increasing demand for neural network applications has led to the development of various machine learning algorithms and their implementation on multiple platforms. Central Processing Units (CPUs), the versatile workhorses of general-purpose computing, feature multiple cores and support for SIMD (Single Instruction, Multiple Data) instructions, enhancing their suitability for diverse tasks and hybrid workloads. While CPUs can offer lower latency, they generally have fewer cores compared to more specialised hardware platforms, limiting their performance in highly parallelisable tasks. General-purpose CPUs are not traditionally designed for parallelism and are therefore less suitable for complex tasks involving intensive neural network computations compared to specialised hardware platforms for neural network acceleration.

Several hardware platforms exist for neural network acceleration, designed to execute specific tasks more efficiently than general-purpose CPUs. In the context of deep learning, these hardware platforms handle the intensive computations required by neural networks, such as matrix multiplications and tensor operations, thereby improving both speed and energy efficiency. The demand for efficient hardware platforms for neural network acceleration stems from the necessity to process large datasets and complex models swiftly. Traditional CPUs are not optimised for the parallel nature of deep learning tasks, leading to inefficiencies in computation time and power consumption. Hardware acceleration platforms, on the other hand, are tailored to manage these tasks, providing significant improvements in performance and energy efficiency [53].

There are various types of hardware platforms for neural network task acceleration, each with unique characteristics and optimisations for specific tasks. Common types include Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), Tensor Processing Units (TPUs), and Data Processing Units (DPUs). Each type has its strengths and weaknesses, making them suitable for different applications and stages of the

machine learning pipeline [54].

Among the primary hardware devices for efficient neural network training and execution are GPUs, reconfigurable hardware like FPGAs [55], and Edge Artificial Intelligence (EdgeAI) devices. GPUs are the most commonly used for inference due to their flexibility and the availability of software frameworks that simplify training, design, and inference. Despite their significant computational capacity, GPUs may be surpassed in speed and energy efficiency by FPGA-based neural network inference acceleration platforms, which integrate both software and hardware optimisation techniques [56]. FPGAs and ASICs are generally more efficient than general-purpose devices such as GPUs and are mainly used for specific neural network implementations. EdgeAI devices, integrated with a host device, manage neural network execution with dedicated hardware platforms designed to process specific operations, such as convolutions, improving efficiency and reducing power consumption and cost.

Recently, neuromorphic chips, which mimic the human brain using analogue devices, have demonstrated promising performance and efficiency based on tests measuring single event upset (SEU) and total ionising dose (TID) [57] [58] [59] [60].

2.2.1 Graphics Processing Units (GPUs)

Graphics Processing Units (GPUs) have revolutionised the domain of parallel computing. Originally designed to accelerate image rasterisation, a task inefficient on sequential CPUs, GPUs possess a massive number of cores that can execute thousands of threads simultaneously, making them ideal for the parallelisable nature of neural network operations. Over time, GPUs evolved from being dedicated to gaming, graphics, and video rendering to flexible hardware platforms for high-performance computing (HPC) and safety-critical applications, such as autonomous vehicles. The introduction of general-purpose programming languages for GPUs, such as Open Computing Language (OpenCL) and Compute Unified Device Architecture (CUDA), enables programmers to leverage GPU parallelism for computation. GPUs excel at handling the large-scale parallelism inherent in neural network operations, such as matrix multiplications and convolutions. Modern GPUs consist of various computing units, called streaming multiprocessors (SMs) in CUDA architecture, each capable of executing several threads in parallel. Figure 2.4 provides a simplified architecture of modern GPUs composed of an array of SMs sharing L2 cache.

Each SM contains basic computing units (CUDA cores in NVIDIA devices) that execute one thread with dedicated registers, avoiding complex resource sharing or long pipelines. Thousands of CUDA cores in a GPU have hundreds of functional units of varying precision and tensor cores that speed up convolutions. Instruction and data caches are shared among all active parallel processes in the SM. Robust software ecosystems, including CUDA and Radeon Open Compute (ROCm), provide extensive libraries and tools optimised for neural network training and inference. The programmer must divide the parallel threads into a grid of blocks when designing a kernel. The block size and grid size are specified when launching a CUDA kernel on a GPU. The number of blocks assigned to an SM depends on the number of registers, shared memory in the SM, and resources required by each block. Some blocks will be queued if the grid size exceeds the number of blocks that

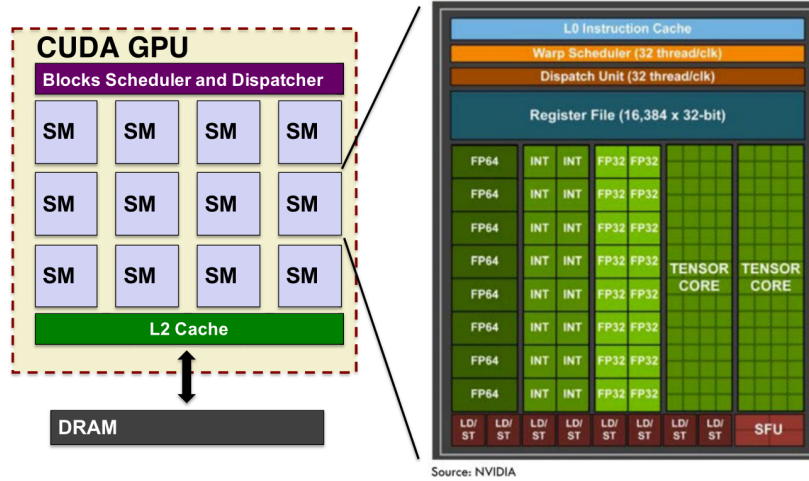


Figure 2.4: Modern GPUs architecture, a simplified view. Adapted from [11].

can be dispatched among the available SMs. Each SM has computing cores with different precisions, based on NVIDIA CUDA devices but applicable to any GPU architecture.

The GPU’s block scheduler dispatches queued blocks to available SMs, reading input data from DDR memories and triggering synchronised block execution. GPUs allow each SM to execute warps (groups) of hundreds of parallel threads in a single computing cycle. If the block size exceeds the available CUDA cores, some threads are delayed until preceding warps are completed. The next block is assigned to the SM only when all threads in the current block are processed. Increasing block size or grid size can improve throughput but must be balanced to optimise parallel capabilities. However, GPUs typically consume significant amounts of power, which can be a constraint in power-sensitive applications such as mobile or edge devices. Additionally, high-performance GPUs can be expensive, making them potentially prohibitive for certain applications. Although GPUs excel in throughput-oriented tasks, they may introduce latency in real-time or low-latency applications. Among the various GPUs available, Nvidia GPUs are widely recognised for their performance and are commonly used in many high-demand computing tasks. These GPUs are known for their advanced architectures and capabilities in accelerating machine learning and neural network computations.

NVIDIA GPUs

NVIDIA GPUs have been the cornerstone of deep learning acceleration due to their parallel processing capabilities. The CUDA programming model allows developers to write highly efficient parallel code, making GPUs versatile tools for both training and inference. NVIDIA GPUs are designed with a large number of cores to perform many calculations simultaneously. Key components include Streaming Multiprocessors (SMs), Tensor Cores, and Memory Hierarchy. SMs are the basic execution units that handle parallel threads. Tensor Cores are specialised cores designed for accelerating deep learning tasks, especially matrix multiplications. The Memory Hierarchy includes global, shared, and cache memories to efficiently manage data flow within the GPU [53] [54].

NVIDIA GPUs are known for their high throughput and versatility, suitable for

both training and inference. They are highly flexible, supporting a wide range of applications and scalable across various platforms. NVIDIA GPUs offer extensive support for deep learning frameworks such as TensorFlow, PyTorch, and MXNet, which simplifies development. NVIDIA GPUs are widely used across various domains, including autonomous vehicles for real-time processing of sensor data for navigation and decision-making, healthcare for accelerating medical imaging and analysis for faster diagnosis, and scientific research for simulating complex physical phenomena and analysing large datasets. However, their high power consumption and cost can be limiting factors, especially for deployment in resource-constrained environments.

2.2.2 Edge Artificial Intelligence (Edge AI)

Edge Artificial Intelligence (Edge AI) devices, such as Google’s Edge TPU and NVIDIA’s Jetson series, are designed to bring AI capabilities to the edge of the network, enabling local data processing and inference without relying on cloud-based resources. Certain EdgeAI devices, such as NeuroShield or Google Coral TPUs, serve as low-cost hardware platforms for CNN execution. They perform basic operations like convolutions and other matrix operations in low precision, significantly reducing time and power consumption for CNNs. Figure 2.5 illustrates the Coral TPU architecture, composed of a hardwired systolic array fed by input buffers, outputting model weights and layer inputs into the activation unit. The Coral TPU for example, performs specific operations efficiently, achieving 2 tera operations per second (TOPS) per watt.

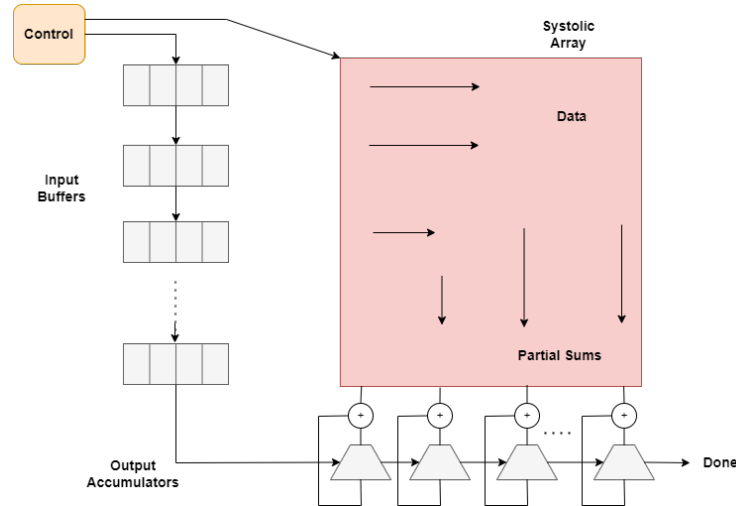


Figure 2.5: Coral Edge TPU architecture [61].

Data within the TPU is represented as 8-bit unsigned integers (UINT8) to minimise transfers and storage and speed up calculations. The device performs quantisation and dequantisation steps to interface with host floating-point representations. The Coral TPU, as an hardware platforms for neural networks acceleration, must be connected to a host device. Google provides versions interfacing via PCIe and USB 3.0. The Coral TPU’s software layer is based on TensorFlow Lite, a light version of TensorFlow optimised for embedded devices. An EdgeTPU compiler deploys the TensorFlow Lite model targeting the Coral Edge TPU architecture. By processing

data locally, edge AI devices significantly reduce latency, making them suitable for real-time applications. Reducing the need to transmit large volumes of data to and from the cloud conserves bandwidth and enhances privacy [62]. These devices are typically compact and energy-efficient, making them ideal for deployment in mobile and portable applications. However, edge AI devices generally offer lower computational power compared to data centre-grade GPUs or CPUs. Limited resources on edge devices can constrain the complexity and size of deployable neural network models. While suitable for small to medium-scale deployments, edge AI devices may face challenges in scaling up for large-scale applications.

Google TPU

Google’s Tensor Processing Unit (TPU) is a custom ASIC designed specifically for accelerating deep learning workloads. It has been used extensively in Google’s data centres to power various AI applications. The TPU architecture is optimised for matrix multiplications, a core operation in deep learning. Key components include the Matrix Multiply Unit (MXU), Unified Buffer, and Vector Processing Units (VPUs). The MXU is the heart of the TPU, designed to perform high-throughput matrix multiplications. The Unified Buffer stores model parameters and intermediate results, facilitating data reuse. The VPU handles vector operations such as activation functions and normalisation [63].

TPUs are designed to deliver high performance for deep learning tasks, significantly outperforming traditional CPUs and GPUs. They can be scaled across large data centres, enabling efficient training and inference for massive models. TPUs are also optimised for power efficiency, reducing the energy cost of running AI workloads. TPUs are used in a wide range of applications, including search and recommendation, powering search algorithms and recommendation systems for large-scale web services, natural language processing, accelerating NLP models for tasks such as translation and sentiment analysis, and image recognition, enhancing image and video recognition capabilities for various applications.

Intel Nervana NNP

Intel Nervana Neural Network Processors (NNP) are specialised chips designed to accelerate deep learning workloads. They focus on high-throughput training and inference, leveraging Intel’s expertise in processor design. Intel NNPs are built with several key architectural features, including Tensor Processing Clusters (TPCs), On-Chip Memory, and Scalable Fabric. TPCs handle tensor operations with high parallelism. On-Chip Memory is designed to reduce latency and increase throughput by keeping data close to the processing units. The Scalable Fabric allows multiple NNPs to be connected, scaling the computational power for large models [63].

Intel NNPs are optimised for training with high energy efficiency. They emphasise energy efficiency while maintaining performance for training tasks. Intel NNPs can be scaled to handle larger models and more data. Intel NNPs are used in various applications, including data centres for accelerating AI workloads in cloud environments, healthcare for enhancing the performance of AI-driven medical diagnostics and treatment planning, and financial services for improving the speed and accuracy of financial predictions and analysis.

Habana Labs Gaudi

Habana Labs, acquired by Intel, developed the Gaudi AI processors designed specifically for deep learning training. Gaudi aims to provide high efficiency and scalability for training large-scale neural networks. The Gaudi architecture includes Tensor Processing Cores, High-Bandwidth Memory, and a Scalable Architecture. Tensor Processing Cores are optimised for high-throughput tensor operations. High-Bandwidth Memory ensures fast data access and transfer rates. The Scalable Architecture supports interconnecting multiple Gaudi processors to scale performance [53].

Gaudi processors are designed to deliver high performance for training large neural networks. They are optimised for power efficiency to reduce the operational costs of training. Gaudi processors allow for the scaling of training resources by interconnecting multiple Gaudi processors. Gaudi processors are used in large-scale training for efficiently training large neural networks for various applications, cloud AI for providing scalable and efficient AI training solutions in cloud environments, and enterprise AI for enhancing AI capabilities in enterprise applications, including analytics and automation.

2.2.3 Field-Programmable Gate Arrays (FPGAs)

Field-Programmable Gate Arrays (FPGAs) offer a unique blend of flexibility and performance, making them suitable for neural network inference in various applications. These reconfigurable devices can be programmed to implement custom hardware platforms tailored to specific applications, including neural network acceleration. FPGAs are integrated circuits designed to be configured by the customer or designer after manufacturing, allowing them to be reprogrammed for different tasks. This reconfigurability makes FPGAs highly versatile for various applications, including neural network deployment, where optimisation of hardware resources based on the specific requirements of the network is crucial.

Deploying neural networks on FPGAs involves mapping the computational graph of the network onto the hardware resources of the FPGA. This process includes designing custom hardware blocks to implement the various layers of the network, such as convolutional and fully connected layers. Additionally, optimising data flow and memory access patterns is essential to ensure efficient execution.

There are two primary architectures for neural network deployment on FPGAs: streaming architectures and single computation engines [64]. Streaming architectures assign each layer of the network to a distinct hardware block, enabling concurrent execution and high efficiency. However, this approach requires generating a new bitstream for each network, which can be time-consuming. In contrast, single computation engines use a fixed architecture with processing elements that execute the layers sequentially. This method allows for reusing the same bitstream across different networks but may sacrifice some efficiency compared to streaming architectures [64].

Key components of FPGAs include configurable logic blocks, interconnects, and memory elements. FPGAs allow users to define executable circuits using these components, enabling high parallelism and efficient mapping of neural networks. Figure 2.6 provides a simplified architecture of FPGAs, illustrating the programmable logic blocks, memory, and interconnections. The high degree of parallelism and number

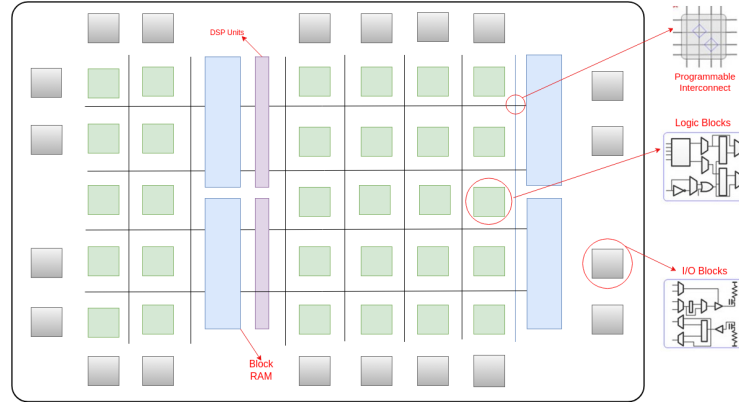


Figure 2.6: FPGAs architecture, a simplified view

of connections make FPGAs particularly efficient for neural network tasks. FPGAs are also known for their flexibility and low latency, making them suitable for applications requiring real-time processing and custom hardware configurations. They are used in various domains, including telecommunications for accelerating network functions, automotive for advanced driver-assistance systems, and data centres for efficient processing of machine learning workloads. Their ability to be reprogrammed also makes them ideal for prototyping and research in deep learning.

However, designing neural network architectures on FPGAs presents challenges related to size and resource availability. For instance, implementing a neuron function typically involves a sigmoid function described using fixed-point data in a Hardware Description Language (HDL). While this approach is complex, it is necessary for efficient FPGA utilisation. Researchers have explored various techniques to reduce FPGA resource usage, including data precision reduction, operation complexity minimisation, weight trimming, and weight quantisation. Binary neural networks (BNNs), which constrain weights to -1, 1, can also reduce resource utilisation, though this may come at the cost of decreased model accuracy [11].

Large-scale deep learning architectures demand extensive parallel processing, memory resources, high throughput, and substantial processing power. This is particularly challenging for safety-critical applications that require real-time hardware systems, where a balance between performance, energy efficiency, rapid development, and cost is essential [55]. Implementing a full CNN or ANN on an FPGA involves numerous connections and processing elements (PEs). Systolic arrays, networks of PEs that perform rhythmic computations, have been utilised to enhance the efficiency of matrix multiplication operations common in CNNs.

FPGA-based platforms have gained attention for their superior performance, high energy efficiency, and exceptional flexibility [65], making them particularly valuable in applications where low latency, high throughput, and energy efficiency are crucial. Despite their advantages, adapting a neural network to an FPGA can be complex and may deter users accustomed to GPU frameworks. To address this complexity, FPGA vendors are developing High-Level Synthesis (HLS) tools compatible with machine learning frameworks. These tools simplify the implementation process, although they may reduce control over the final circuit and complicate reliability evaluation. Nevertheless, FPGAs offer significant performance improvements by allowing the creation of highly optimised custom circuits for specific neural network architectures. They provide deterministic low-latency processing, crucial for real-

time applications, and can achieve high energy efficiency by optimising the hardware for the target workload.

However, developing FPGA-based solutions demands expertise in HLS, HDLs and a deep understanding of hardware architecture. The design and verification process for FPGA-based solutions can be more time-consuming compared to software-based solutions on GPUs or CPUs. While FPGAs present a powerful option for neural network acceleration, particularly in applications requiring real-time processing and energy efficiency, their development and implementation pose significant challenges. Furthermore, although FPGAs provide a robust platform for deep learning, optimising their resources is crucial to efficiently manage memory bandwidth and achieve faster inference. Despite advancements in FPGA-based neural network deployment, challenges remain, particularly in ensuring predictability and reliability in safety-critical applications. These systems require stringent guarantees on timing and behaviour, necessitating advanced techniques to manage runtime unpredictability and resource constraints.

Comparative Analysis of Hardware Platforms for Neural Networks Deployment

The increasing demand for applications of CNNs and ANNs has driven the development and implementation of various machine learning algorithms across different hardware platforms, including GPUs, CPUs, and reconfigurable hardware such as FPGAs [66]. These neural networks, especially large-scale ones, require substantial computational power, significant memory resources, and considerable storage, which can limit their applicability depending on the hardware platform used.

GPUs are often the preferred platform for deploying neural networks due to their high computational capacity and user-friendly development frameworks. They offer high throughput and versatility, making them suitable for both training and inference tasks. CPUs, while offering lower computational power compared to GPUs, are still widely used for hybrid workloads where deep learning tasks are integrated with other types of processing. However, their lower parallelism makes them less efficient for large-scale neural network tasks.

FPGAs present a promising alternative for neural network inference acceleration by combining software and hardware optimisation techniques. These platforms can potentially surpass GPUs in terms of speed and energy efficiency [67]. FPGAs provide flexibility in customising hardware configurations, which is particularly valuable in applications where low latency, high throughput, and energy efficiency are critical.

When evaluating the suitability of these hardware platforms for neural network applications, several factors must be considered, including performance, power efficiency, development complexity, and cost. The choice of hardware acceleration platform is typically influenced by the specific requirements of the application, some represented on Table 2.1.

A comparative analysis of these hardware platforms reveals their unique strengths and suitability for different applications:

- **NVIDIA GPUs** are known for their high throughput and flexibility, supporting a wide range of deep learning frameworks and tools, making them highly accessible for both training and inference [54].

Platform	Performance	Rate Power Efficiency	Development Complexity	Cost
GPU	High	Moderate	Moderate	High
CPU	Moderate	Low	Low	Low
FPGA	High	High	High	Moderate
Edge AI	Low / Moderate	High	Moderate	Low / Moderate

Table 2.1: Comparative analysis of hardware platforms.

- **Google TPUs** excel in high-throughput inference tasks, particularly in large-scale deployments. They are designed for high performance with lower energy consumption compared to traditional GPUs, making them efficient for specific AI workloads [63].
- **FPGAs** offer high flexibility and customisation, allowing for the implementation of tailored neural network architectures. These reconfigurable circuits are especially useful in scenarios requiring deterministic low-latency processing and energy efficiency, such as real-time applications in medical diagnostics and imaging.
- **Intel NNPs (Neural Network Processors)** are optimised for training tasks, emphasising energy efficiency while maintaining strong performance. They provide a balance between performance and power consumption [54].
- **Habana Gaudi** targets large-scale deep learning workloads in data centre environments, focusing on scalable training with high efficiency and performance [53].

In terms of energy efficiency, TPUs and FPGAs are designed to deliver high performance with lower energy consumption. TPUs are optimised for specific AI workloads, while FPGAs can be reconfigured to maximise energy efficiency for particular neural network architectures.

Regarding flexibility and scalability, FPGAs stand out for their reconfigurability, which allows them to adapt to evolving neural network models and application-specific requirements. NVIDIA GPUs also provide significant flexibility and scalability across various platforms, making them versatile tools in diverse environments. Google TPUs, meanwhile, are optimised for scalability in data centre environments and integrate seamlessly with TensorFlow, offering a streamlined experience for developers.

Finally, the developer support and user-friendliness of these platforms vary. NVIDIA GPUs are widely supported by deep learning frameworks and tools, making them user-friendly for developers. FPGAs, while offering extensive customisation options, require more specialised knowledge and tools, though vendors like Xilinx provide APIs and tools aimed at simplifying deployment. Google TPUs, integrated with TensorFlow, are designed to be user-friendly for those already working within that ecosystem.

The choice of platform often depends on the specific requirements of the application. For instance, medical applications, particularly those involving neural network inference for diagnostics and imaging, demand high accuracy, low latency,

and energy efficiency. A comprehensive analysis of GPUs, CPUs, FPGAs, and edge AI devices shows that FPGAs stand out as the most suitable option for applications requiring portability, high performance, and power efficiency at a moderate to low cost. Their ability to provide customised, high-performance, and energy-efficient solutions creates the potential to adapt accelerators to achieve the stringent requirements of safety-critical applications. This adaptability makes FPGAs particularly valuable for safety-critical applications, where the demands for reliability and precision are paramount. Consequently, this dissertation leverages the inherent flexibility of FPGAs, adapting them for customisation and reconfiguration to meet the specific needs of medical diagnostics and imaging. Although FPGA-based solutions may involve a complex and time-consuming development process, their long-term benefits in performance, power efficiency, and reliability, which contribute to achieving higher throughput, make them especially well-suited for neural network acceleration [32] and the preferred choice for safety-critical medical applications.

2.3 Deep Learning Compilers

The use of Deep Learning model computations such as Math Kernel Library for Deep Neural Networks (MKL-DNN) and CUDA Deep Neural Network library (cuDNN) on general-purpose hardware is facilitated by a variety of tools, ranging from Basic Linear Algebra Subprograms (BLAS) libraries to specialised hardware developer libraries. These include pooling, convolution, and normalisation libraries, as well as more advanced operators such as TensorRT [68], which optimises computation graphs [69]. Despite the availability of these tools, utilising Deep Learning frameworks to directly access hardware tools and libraries presents significant challenges due to hardware diversity. This diversity results in inefficient mapping capabilities of Deep Learning frameworks to hardware, leading to suboptimal hardware resource utilisation [70].

In the context of deep learning, the deployment of neural network models necessitates the use of efficient and flexible neural network compilers. Several Deep Learning compilers have been introduced to tackle the challenges associated with the manual optimisation of Deep Learning models for specific hardware. These compilers load inputs from Deep Learning frameworks and execute models using optimised code generation techniques such as layer and operator fusion, based on hardware intrinsics. Most Deep Learning compilers leverage existing toolchains such as Low Level Virtual Machine (LLVM) and NVIDIA CUDA Compiler (NVCC) to enhance portability and efficiency during execution, enabling them to target a broad range of hardware platforms effectively [71].

Several compilers have been developed to address these needs, among the most widely used and recognised compilers are nGraph [72] by Intel, Glow [73] by Facebook, TVM [74] by Apache, and XLA [75] by Google. This dissertation briefly evaluates and compares these four popular Deep Learning compilers - Apache TVM, Google XLA, Facebook Glow, and Intel nGraph - focusing on their functional components within the layered-based design architecture, including frontend support, intermediate representation, and backend support. The compilers are assessed based on their capabilities and suitability for customisable applications.

The study evaluation on Table 2.2 indicates that Apache TVM stands out for customisable applications, mainly due to its superior FPGA hardware backend support

Compiler	Frameworks	Data Types	Support	Hardware	Optimisation
Apache TVM	TensorFlow, Keras, PyTorch, MxNet, Caffe, Tflite, ONNX, CoreML, DarkNet	Dynamic Shapes, 8-bit, 16-bit	Open-Source, recustomize, Halide Integration	CPU, ARM, FPGA, GPU, VTA	Layer Fusion, Auto-tuning, Quantisation
Google XLA	TensorFlow	8-bit, 16-bit	HLO for IR, Non-Dynamic Shapes	CPU, GPU, TPU	Operator Fusion, JIT Compilation
Facebook Glow	PyTorch, Caffe2, TensorFlow Lite	8-bit, Non-Dynamic Shapes	Built-in ONNXIFI, Static Shapes	CPU, GPU	Lowering, Operator Fusion, Quantisation
Intel nGraph	TensorFlow, PaddlePaddle	8-bit, Partial Shapes	IR Transformation, Kernel Libraries Integration	CPU, Intel GPU, NNP, OpenCL	Layer Fusion, Auto-tuning, Quantisation

Table 2.2: Comparison of Deep Learning Compilers

compared to other compilers. Additionally, TVM offers quantisation support for 8-bit integer and 16-bit floating-point operations, facilitating FPGA targeting. TVM’s low-level intermediate representation optimisation methods, which utilise machine learning techniques, are highly efficient, and its support for dynamic shapes provides flexibility for artificial neural network models, despite its primary design for CNNs. TVM is extensively used in various AI applications and delivers high-performance deep learning inference on FPGAs.

The deep learning compilation landscape is vast and complex, with each compiler offering unique features and optimisations. For instance, Apache TVM’s support for a wide range of frameworks makes it highly adaptable and versatile, allowing it to be integrated into numerous deep learning workflows. Leveraging on TVM’s adaptability makes it flexible to deploy models on different hardware platforms without rewriting significant portions of their code.

Google’s XLA, on the other hand, is designed to optimise TensorFlow computations, leveraging its HLO for efficient low-level intermediate representation. While it does not support dynamic shapes, its ability to work seamlessly with TensorFlow makes it an excellent choice for users who are heavily invested in the TensorFlow ecosystem. XLA’s optimisation capabilities can significantly improve the performance of TensorFlow models on supported hardware.

Facebook Glow’s focus on interoperability with ONNXIFI and support for PyTorch and Caffe2 highlights its aim to provide a streamlined compilation process for popular frameworks. Although it lacks support for dynamic shapes, Glow’s efficient

lowering process and compatibility with various hardware backends make it a competitive option for specific use cases. Its design prioritises ease of integration and performance, particularly for inference tasks on CPUs and GPUs.

Intel’s nGraph stands out with its integration of TensorFlow Paddle and support for partial shapes. By leveraging kernel libraries such as cuDNN, nGraph aims to optimise deep learning computations on Intel hardware. Its deployment capabilities across CPUs, Intel GPUs, and NNP GPUs, combined with OpenCL support, make it a robust choice for users looking to maximize performance on Intel architectures.

The choice of a deep learning compiler therefore depends heavily on the specific requirements of the application, the hardware in use, and the frameworks being deployed. Apache TVM’s comprehensive framework support and hardware flexibility make it a standout choice for many applications, especially those requiring dynamic shape support and high-performance inference. Google’s XLA and Facebook Glow offer strong alternatives with their own unique advantages, particularly in terms of ecosystem integration and optimisation capabilities. Intel’s nGraph provides a tailored solution for Intel hardware, making it a valuable tool for maximising performance on these platforms.

Ultimately, the selection of a deep learning compiler is a critical decision that can significantly impact the efficiency and performance of neural network deployments. It is therefore imperative to carefully evaluate the features and capabilities of each compiler to ensure that deep learning models are optimised for the specific use cases, leading to more efficient and effective AI applications. For specific application required in this dissertation, Apache TVM, along with its hardware accelerator VTA, stands out as the preferred solution for hardware acceleration customisation. This preference is due to its open-source nature, dynamic hardware backend support, and the capability to handle dynamic shapes, resulting in high-performance deep learning inference across various hardware platforms [2].

Apache Tensor Virtual Machine (TVM)

Apache TVM is an open-source machine learning compiler framework [76] that optimises code to deploy models on specific hardware, including FPGAs. It accepts inputs from deep learning frameworks such as TensorFlow and PyTorch, supporting various AI applications with high-performance and efficient deep learning inference on FPGAs.

Apache TVM [74] can execute quantised models across different hardware platforms. It enables the quantisation and efficient execution of models from frameworks such as PyTorch, TFLite, TensorFlow, and MXNet with minimal accuracy loss. Utilising 8-bit integer quantisation, it optimises the resources of the target hardware, initially designed for 32-bit floating-point computations, by lowering them to 8-bit integers.

Apache TVM’s architecture, as depicted in Figure 2.7, facilitates the transformation of high-level deep learning models into deployable modules for target devices like FPGAs. The compilation process begins with an import request, bringing the model through the frontend to create an Intermediate Representative (IR) module (IRModule). This IRModule represents the model’s structure as a collection of functions. The compiler then transforms the IRModule to suit the target device, translating it into a usable format for execution. The resulting codegen output is

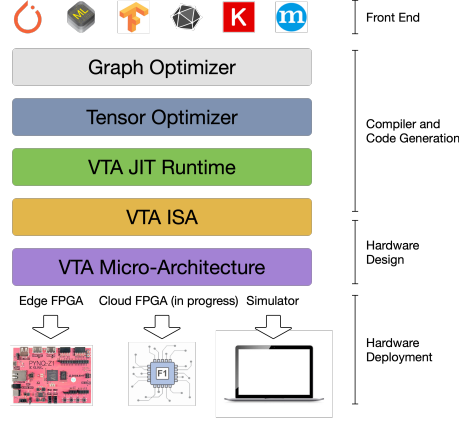


Figure 2.7: TVM Compiler Stack [77].

exported, loaded, and executed as a runtime-Module on the FPGA runtime environment. This runtime-Module is reloaded into the compiler for FPGA runtime execution.

The main data structures and APIs in TVM, shown in Figure 2.8, include the IRModule and the runtime-module. The IRModule supports both the Relay and Tensor Intermediate Representation (TIR) APIs, represented as relay function and TIR primitive function, respectively. The relay function represents the model as a high-level functional program, supporting features such as recursion, control flow, and complex data structures. The TIR primitive function corresponds to a lower-level representation of the IRModule, encompassing multi-dimensional load and store operations, vectorised instructions, and tensor manipulations.

The runtime-module includes the runtime-packed function API for transformations to optimise and lower the program to a target-interpretable lower-level representation. The relay/transform optimises the model by eliminating dead code and performing constant folding [71], among other steps, to streamline tensor computation. The TVM compiler pipeline then passes the relay compute graph through fuse-ops passes [78], reorganising it into fused convolutional layers for efficient sub-function compilation and optimisation.

The unified runtime-module node facilitates various modes of execution. For simple models like ANNs, the relay graph can be directly lowered into a graph executor, while TVM supports dynamic executions for more complex models requiring external code generators. The TIR transformation focuses on optimisation processes like dead code elimination [79] and index simplification [80]. However, most low-level optimisations are managed by CUDA or LLVM at the target phase. Key TIR functions include rolling out one-dimensional access from multi-dimensional access and providing specific configurations for the target hardware.

TVM also performs optimisations such as access to multi-dimensional tensors, visual threading [74], and loop tiling and fusion [81], supporting high-performance code optimisation for various targets. Additionally, scheduling primitives provide a machine-learning algorithm that guides scheduling sequences to achieve the best performance. Pre-trained deep learning models are imported and converted to a TVM-compatible format using the IRModule function. TVM then generates a computation graph and constructs a dataflow graph from the model, converting it to a Relay IR graph. This high-level IR is optimised through various passes, includ-

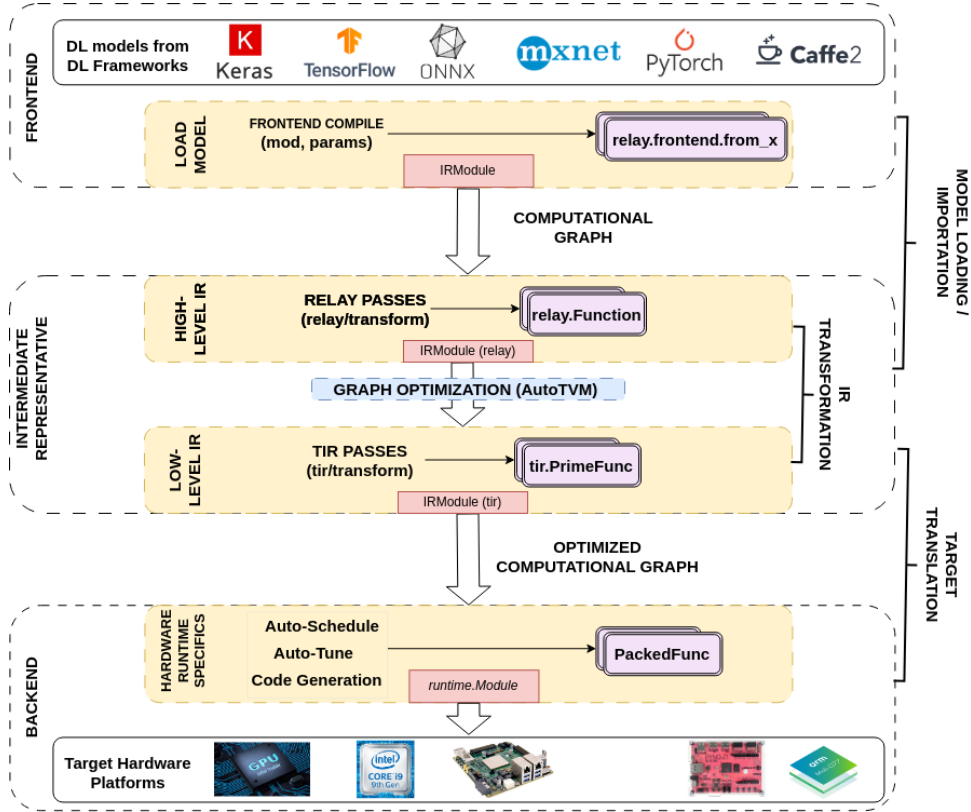


Figure 2.8: TVM IP Structure Framework

ing operator fusion and constant folding, before being converted to a TVM graph format. The optimised TVM graph is further compiled for the target hardware platform.

2.4 Common Deep Learning Hardware and Inference Acceleration Frameworks

Deep learning has revolutionised numerous fields by enabling machines to perform complex tasks such as image recognition, natural language processing, and autonomous driving. Specialised hardware accelerators and software frameworks have been developed to achieve the computational power required for deep learning. Common examples include closed-source platforms, such as those paired with inference frameworks like Vitis AI, and open-source hardware accelerators like the Versatile Tensor Accelerator (VTA). Other tools, such as Fast, Scalable Quantised Neural Network Inference (FINN) and High-Level Synthesis for Machine Learning (HLS4ML), are FPGA-based frameworks that provide customisation and deployment solutions for machine learning workloads. This section explores the architectures and functionalities of these frameworks and hardware accelerators, highlighting their critical role in advancing machine learning technologies for AI inference.

2.4.1 Apache TVM-VTA (Versatile Tensor Accelerator)

As already established in Section 2.3, Apache TVM is an open-source deep learning compiler stack that enables high-performance machine learning on a variety of hardware platforms. The Versatile Tensor Accelerator (VTA) is an extension of the TVM framework, designed to provide a programmable accelerator for deep learning [77].

VTA exposes a RISC-like programming abstraction, allowing developers to describe and compute memory operations at the tensor level. It consists of several modules: Fetch, Compute, Load, and Store. VTA is designed to be programmable, allowing users to describe tensor operations using high-level abstractions. It integrates seamlessly with the TVM stack, providing an end-to-end solution for deploying deep learning models on custom hardware. Users can modify VTA’s architectural parameters to suit specific applications and performance requirements. VTA is suitable for a variety of applications, including research, enabling hardware and compiler researchers to experiment with new optimisations and co-design techniques, and prototyping, providing a platform for fast prototyping of deep learning accelerators on low-cost FPGA hardware with lower energy consumption [32].

2.4.2 Vitis Artificial Intelligence (Vitis AI)

Vitis Artificial Intelligence (Vitis AI) is a comprehensive AI inference development platform from Xilinx, designed to optimise and deploy deep learning models on Xilinx hardware platforms, including FPGAs and SoCs. Vitis AI’s architecture comprises several key components that work together to deliver high-performance inference: the Vitis AI Optimiser, Vitis AI Library, and Vitis AI Runtime. The Vitis AI Optimiser performs quantisation and pruning to improve the efficiency of neural network models on Xilinx hardware. The Vitis AI Library provides a set of pre-optimised AI models and applications, while the Vitis AI Runtime offers APIs for model deployment and execution on Xilinx devices [82].

Although not a tensor accelerator, Vitis AI provides key software tools for enhancing deep learning inference on Xilinx hardware by reducing computational complexity and improving model efficiency. It supports a wide range of AI applications, including autonomous driving, medical imaging, and retail analytics, providing tools that can scale across different hardware platforms, from edge devices to data centres.

2.4.3 Fast, Scalable Quantised Neural Network Inference (FINN)

FINN is an open-source framework from Xilinx Research designed to build and deploy FPGA-based DNN accelerators, explicitly focusing on quantised neural networks (QNNs). FINN is tailored for low-bandwidth networks, which allow for reduced hardware resource usage while maintaining high throughput and low latency. While FINN itself is not a hardware accelerator, it provides a framework for generating custom FPGA accelerators for deep learning inference [83].

FINN is utilised in applications requiring low-power, high-performance inference at the edge, such as IoT devices, autonomous systems, and industrial automation. It allows developers to rapidly prototype and deploy custom QNN accelerators, optimised for specific use cases that demand power and resource usage efficiency.

2.4.4 High-Level Synthesis for Machine Learning (HLS4ML)

HLS4ML (High-Level Synthesis for Machine Learning) [84] is an open-source tool that enables the conversion of machine learning models into FPGA-optimised implementations. HLS4ML leverages high-level synthesis (HLS) to generate low-latency, high-throughput FPGA accelerators from machine learning models. It supports various machine learning frameworks and provides tools for converting and optimising models for deployment on FPGA hardware [85].

HLS4ML is particularly suited for scientific research, including real-time data processing in particle physics and astrophysics, where low latency is critical. It is also applied in edge computing and embedded AI applications, enabling the deployment of compact, efficient models on resource-constrained devices.

2.4.5 Versatile Tensor Accelerator (VTA)

In this dissertation, the initial focus is on adapting an existing tensor accelerator, the Versatile Tensor Accelerator (VTA), to enhance predictability and improve memory access efficiency. The VTA is an open-source hardware accelerator developed as part of the Apache TVM framework, designed specifically to optimise the deployment of deep learning CNN models on FPGAs. By offering a streamlined, end-to-end solution, VTA facilitates efficient model deployment through its modular and open-source architecture, which can be easily implemented on FPGA platforms. Additionally, VTA provides a simulator for prototyping on standard workstations, making it accessible for various development environments. The VTA software stack includes a driver and just-in-time (JIT) runtime that supports both simulators and FPGA hardware back-ends, seamlessly integrating with the TVM compiler stack to optimise and deploy models from diverse deep learning frameworks [86].

The VTA hardware and software stack supports quantisation methods through its SDK and HDL implementations. The VTA SDK offers tools for model quantisation, compilation, and optimisation, while the VTA HDL provides the hardware implementation needed for efficient execution of quantised models on FPGAs. By performing quantisation on the FPGA, VTA optimises deep learning models to leverage the specific hardware resources of the FPGA, enabling efficient execution.

VTA serves as an extension of the Apache TVM framework, offering a programmable accelerator with a RISC-like programming model for defining compute and memory operations at the tensor level. This model reflects common features of mainstream deep learning accelerators, such as tensor operations, DMA load/stores, and compute/memory arbitration. VTA includes a behavioural hardware simulator and supports deployment on low-cost FPGA hardware for rapid prototyping, providing a transparent end-to-end deep learning system from high-level frameworks to hardware implementation.

2.4.5.1 Instruction Set Architecture

VTA’s instruction set architecture (ISA) comprises four complex instructions with varying execution latencies:

- **LOAD:** Transfers a 2D tensor from Dynamic Random-Access Memory (DRAM) to the input buffer, weight buffer, or register file, and can also load a micro-kernel into the micro-op cache with dynamic padding support.

- GEMM: Executes matrix-matrix multiplications on input and weight tensors, adding the results to a register-file tensor.
- ALU: Conducts matrix-matrix ALU operations on register-file tensor data.
- STORE: Transfers a 2D tensor from the output buffer back to DRAM.

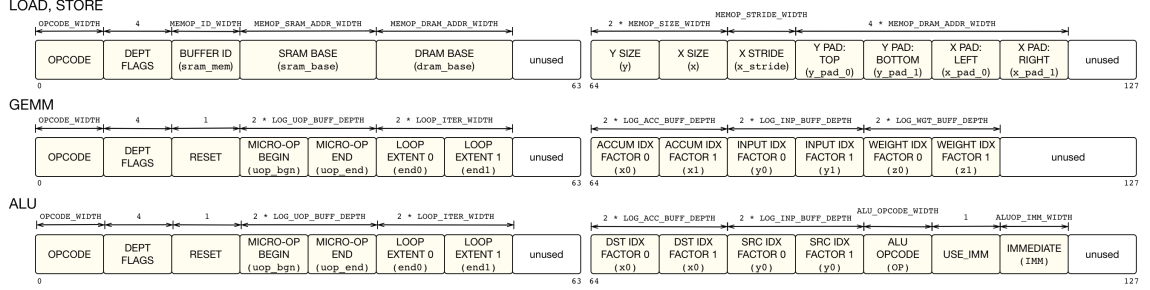


Figure 2.9: VTA Instructions [77].

LOAD instructions are handled by load and compute modules, GEMM and ALU instructions by the compute module’s GEMM core and tensor ALU, respectively, and STORE instructions by the store module. The VTA ISA adjusts based on architectural parameters like GEMM core shape, data type, and memory size, with the VTA runtime generating tailored binary code accordingly.

VTA utilises dependence FIFO queues to synchronise tasks across hardware modules. Modules execute concurrently, communicating through read-after-write (RAW) and write-after-read (WAR) dependence queues. Dependence flags are decoded in hardware, and execution relies on receiving RAW tokens from producer modules and WAR tokens from consumer modules [77]. Upon task completion, outgoing RAW and WAR dependencies notify the respective consumer and producer modules.

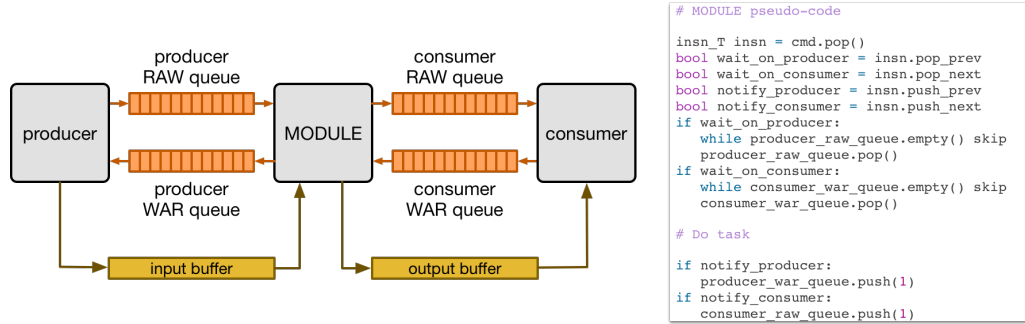


Figure 2.10: VTA Data Flow [77].

The next section describes the detailed structure and data flow within the VTA, highlighting the interconnectedness and communication between the four primary modules to achieve efficient deep learning computation.

2.4.5.2 Versatile Tensor Accelerator (VTA) Structure

The Versatile Tensor Accelerator (VTA) is designed with a modular architecture, consisting of four primary modules: Fetch, Load, Compute, and Store. These modules communicate through FIFO queues and local SRAM blocks, enabling efficient

task-level pipeline parallelism [77]. The following sections detail the functions of each module.

- **Fetch Module:** The Fetch module is responsible for initialising the VTA by loading a stream of instructions from DRAM. It does this by utilising three memory-mapped registers: control, insn_count, and insns. Once the instructions are loaded, the module decodes them and transfers them into command FIFO queues. These queues then direct the instructions to the appropriate modules—Load, Compute, and Store. The Fetch module ensures continuous operation by stalling if any command queue becomes full, thus maintaining a steady flow of tasks throughout the system.

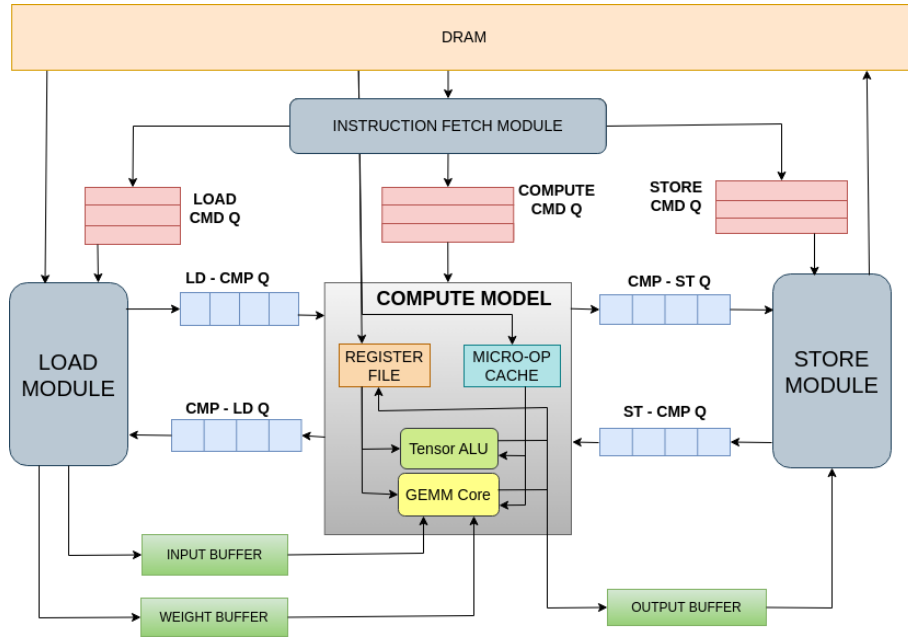


Figure 2.11: VTA Structure [77].

- **Load Module:** The Load module's primary function is to transfer input, weight, and bias tensors from DRAM into dedicated on-chip memory using 2D DMA transfers. It includes on-the-fly padding for 2D convolution inputs and weights, which facilitates efficient tiling and reduces the need for data rearrangement in DRAM. This mechanism ensures that the data is readily available for computations, maintaining a consistent data flow for the Compute module to process.
- **Compute Module:** The Compute module handles the core computational tasks of the VTA. It features a RISC processor with two main components: the tensor ALU for general computations and the GEMM core for dense linear algebra operations. The Compute module executes ALU and GEMM operations from micro-op sequences, leveraging a two-level nested loop structure to optimise the instruction footprint. Additionally, it manages data by loading it from DRAM into the register file of the on-chip memory and loading micro-op kernels into the micro-op cache. This setup allows the module to perform efficient and rapid computations.

- **Store Module:** The Store module is responsible for retrieving computation results from the Compute module and transferring these 2D tensors back to DRAM. This module plays a crucial role in ensuring that the computed data is efficiently stored and synchronised with the Fetch and Compute modules. By doing so, it supports the seamless operation of the entire VTA pipeline, allowing for the smooth execution of subsequent tasks.

2.4.5.3 FPGA Quantisation Processes

FPGA quantization is the process of reducing the bit width of weights and activations in a deep learning model to optimise the model for the specific hardware resources of an FPGA. This process decreases the hardware’s resource usage by sharing parameters and helps in achieving efficient model execution on the target FPGA [87].

The VTA backend supports three main quantisation processes for deploying neural network models on FPGAs:

- **Integer Quantisation:** This method enhances memory throughput and reduces processing time by quantising the weights and activations into integers with fewer bits than their floating-point counterparts. By representing weights and activations as smaller bit-width integers, the model can be executed more efficiently on FPGA hardware.
- **Dynamic Fixed-Point Quantisation:** This technique maintains the model’s accuracy by using a fixed number of bits that can be dynamically modified during inference to represent weights and activations. This flexibility allows for adjustments that can preserve the precision of the computations while optimising the model for the FPGA.
- **Hybrid Quantisation:** A combination of integer and floating-point quantisation, this method quantises activations to floating-point numbers and weights to integers. This hybrid approach seeks to balance the advantages of both quantisation methods, optimising resource usage and maintaining model accuracy.

To optimise models for FPGA-specific hardware constraints and reducing the memory footprint, Apache TVM quantises 32-bit float values to 8-bit integers using the `relay.quantise.quantise` function, reducing memory and computation requirements for efficient execution on FPGAs. Apache TVM leverages VTA to perform a range of quantisation strategies, aiming to strike a balance between model accuracy and FPGA resource constraints to enhance efficiency. The quantisation process is particularly effective for models with fewer layers, as quantisation impacts performance based on the depth of the model [87].

After quantisation, the model deployment workflow for using the VTA backend involves designing and training the model, performing quantisation, compiling the model, configuring the hardware, inputting data, running inference, and post-processing results. The TVM compiler takes the high-level neural network model, translates it into a hardware description language (HDL), performs various optimisations, and generates a bitstream for FPGA deployment. The compiled model is

loaded into the compiler stack to create a runtime module, converting the model into a graph format compatible with the VTA backend.

Inference is performed using the transformed input data on the configured FPGA, programmed with the bitstream. The VTA backend executes the neural network layers, obtains output tensors, and converts them into a readable format for application use. This comprehensive workflow ensures efficient deployment and execution of deep learning models on FPGA hardware using the VTA backend.

2.5 Requirements in Safety-Critical Artificial Intelligence (AI) Applications

The previous chapter discussed the critical importance of reliability in safety-critical domains. Expanding on this, Biondi et al. [88] have identified several challenges associated with the application of deep learning software in safety-critical systems, including issues related to predictability, memory throughput, safety, security, and energy efficiency [88]. Systems operating in these environments must adhere to stringent requirements to prevent catastrophic consequences. The following key properties are typically essential for such systems:

- **Dependability:** Dependability encompasses several attributes, including reliability, availability, and maintainability. The system must operate reliably under predefined conditions, be resilient to errors and failures, and ensure continuous functionality. Reliable systems consistently perform their intended functions without failure. Availability ensures that the system is operational whenever required, and maintainability allows for quick restoration after a failure. Ensuring dependability is crucial in environments where failures can result in significant harm or loss.
- **Time Predictability:** Systems must guarantee that operations are completed within specified time constraints, a critical factor in real-time applications where delays can lead to unsafe outcomes. These systems must respond within predefined time bounds, determined during the design phase based on specific performance requirements. They need to be analysable and verifiable to ensure optimal performance in both functional and temporal domains. For example, even slight deviations in response time could be disastrous, whether in medical devices like implantable defibrillators, where timely detection and intervention are crucial for patient survival, or in the control systems of autonomous vehicles, where rapid and accurate reactions are essential to prevent accidents, time predictability is essential.
- **Memory Management:** Efficient and predictable memory usage is crucial to ensure that the system can handle high workloads without performance degradation, particularly in environments with limited resources. Effective memory management is essential to address issues like memory interference, collisions, and access conflicts, which can lead to unpredictable system behaviour. Safety-critical systems must ensure efficient and reliable memory access to prevent delays and errors. Techniques such as memory partitioning, error correction codes, and predictable memory allocation algorithms are necessary to maintain system stability and performance.

- **Energy Efficiency:** Minimising energy consumption is vital, especially for systems with constrained power supplies, such as embedded systems, where energy efficiency directly impacts operational longevity and reliability. Energy efficiency is critical in safety-critical systems, particularly those deployed in resource-constrained environments. Systems must optimise power consumption to extend operational life and ensure sustainable performance. This is especially important in applications like autonomous vehicles and medical devices, where power constraints are significant.
- **Security:** Systems in safety-critical domains are vulnerable to cyber attacks that could compromise crucial components and make the system susceptible to external manipulation. For instance, in an autonomous defibrillator medical system, a cyber attack could alter the control flow of vital sensors, potentially leading to fatal outcomes by disrupting the delivery of life-saving treatments.
- **Safety and Fault-Tolerance:** The system must be designed to avoid hazards and tolerate faults, ensuring that any failures do not lead to unsafe states or behaviours. This includes mechanisms for detecting, isolating, and mitigating faults to maintain system integrity. Safety-critical systems must incorporate robust mechanisms to detect and tolerate faults to the bare minimum to prevent catastrophic consequences for both personnel and the environment. These systems must be able to handle unexpected failures gracefully.
- **Certifiability:** Systems must meet stringent standards set by certification authorities to be deemed safe for use in critical applications. Achieving certifiability often involves rigorous testing, validation, and documentation to ensure compliance with established safety and performance

Although neural networks have achieved human-level performance in many tasks, integrating them into safety-critical systems poses significant challenges. This is primarily because the inference output of neural networks is not entirely reliable, especially when encountering inputs that differ from their training data, leading to incorrect responses. Notable incidents, such as the Tesla Autopilot crash, exemplify these challenges where system failures led to severe consequences [89]. Moreover, neural networks can learn unintelligible input-output mappings prone to adversarial attacks, which can cause unpredictable behaviour by altering a few input pixels [90].

Addressing these stringent requirements is demanding. Researchers have proposed several solutions, but integrating neural networks into safety-critical systems remains complex. The primary issues stem from the inference processes of neural networks. Many neural network acceleration hardware platforms introduce significant and unpredictable delays, particularly in worst-case scenarios where concurrent operations, such as memory access, interfere with processes or multiple neural networks execute simultaneously on the same platform. These inference engines are typically designed for average performance, not the stringent demands of safety-critical applications [88].

2.5.1 Dependability in Memory Access for Safety-Critical Applications

Dependability in computing systems, particularly in relation to memory access, is a critical factor that ensures the reliable operation of AI and ML applications. Memory access inefficiencies can lead to latency, reduced throughput, and increased power consumption, all of which are detrimental to the performance of neural networks in resource-constrained embedded systems. Addressing these issues is crucial for maintaining the integrity and effectiveness of AI systems deployed in various applications.

AI and ML are increasingly integral to modern applications, emphasising intelligent autonomy. Deploying neural networks in resource-constrained embedded systems poses significant challenges, especially regarding latency, throughput, power consumption, and cost. This is particularly evident when using common commercial off-the-shelf (COTS) embedded edge devices with limited power and computational resources [91].

As neural network models grow in complexity, the computational demands for deploying them on edge devices increase significantly, often surpassing the capabilities of these platforms. Addressing these demands requires efficient, specialised hardware platforms for neural network acceleration that offer excellent performance, dependability, and energy efficiency [91]. This need is essential to ensure that neural networks continue to be relevant across a wide array of applications.

Moreover, the dependability of accelerator-based systems in such applications is critical. These systems often extract valuable information from complex raw data, where failures can have catastrophic consequences, including loss of life or severe environmental damage [92]. The continuous scaling of CMOS technology further exacerbates threats to the dependability of advanced neural network acceleration hardware platforms, introducing risks such as ageing-induced permanent stuck-at faults (PSFs) and timing errors (TEs).

Previous research has focused on enhancing the resilience of neural network hardware platforms against dependability threats. For instance, [93] explored the resilience of hardware platforms for neural network acceleration to fixed, transient, and timing errors induced by thermal effects, process variations, and voltage and memory refresh rate scaling. The study by [94] delved into improving reliability by analysing the inherent fault tolerance of neural network models in different hardware platforms, categorising fault tolerance methods for passive and active faults to propose more robust platforms against transient errors. Another approach examined near-threshold computing (NTC), where [95] presented techniques to manage NTC-induced TEs, evaluating factors such as utilisation patterns, workload characteristics, hardware platform homogeneity, and delay variations within arithmetic units.

Further studies have adopted hierarchical approaches to fault tolerance in deep learning models, addressing model, architecture, circuit, and cross-layer perspectives [96]. Other research analysed intrinsic reliability threats in neural networks and corresponding mitigation techniques [97], while some tackled random and systematic hardware failures and safety certification challenges in complex, parallel, compute-intensive software functions in GPUs [98].

Recent advances in hardware platforms for neural network acceleration for safety-

critical domains focus on optimising data-flow, communication networks, and integrating emerging technologies and computing components [99]. Research has also examined the the nature of neural network models, their development requirements, including hardware and software needs [100], and optimisation techniques for efficient deployment, particularly in resource-constrained edge devices such as FPGAs [101] [102].

Dependability is critical for intelligent edge devices used in safety-critical applications. Studies have leveraged aging-aware DNN acceleration hardware platform architectures with robust algorithms and adaptive techniques to manage aging effects while maintaining reliability and efficiency [103] [104]. However, these methods may reduce resilience due to intrinsic redundancy reduction [91].

Further research has explored the dependability of accelerator-based systems by considering ageing mechanisms in neural network accelerators, focusing on aspects such as reliability and longevity, performance degradation, energy efficiency, thermal management, workload balancing, and design optimisation [91]. These factors can degrade hardware lifespan, decrease performance, increase power consumption, and exacerbate thermal issues. [91] proposed techniques to improve the dependability of neural network accelerators from an ageing perspective, addressing functional resilience, new classifications of timing resilience, and lifetime resilience. Their study presented runtime considerations, techniques, and tools to enhance the dependability of neural network accelerators, emphasising error correction codes and adaptive resource allocation to mitigate ageing impacts and extend hardware lifetime.

2.5.2 Predictability of memory accesses in safety-critical systems

Dynamic Random-Access Memory (DRAM) serve as main memory in various applications; however, dynamic memory allocation is typically avoided in hard real-time applications due to the unpredictable timing behaviour introduced by these allocators [105]. Safety-critical embedded systems with AI functionality are facing increasing computational and memory demands, particularly as AI is adopted in edge-computing applications across multiple safety-critical domains. Given the high memory demand, DRAM is often scarce, and mechanisms to support DRAM oversubscription pose serious concerns about GPU/CPU capacity loss and memory access constraints [106], jeopardising its application in safety-critical systems.

According to [107], current dynamic memory allocators introduce unpredictability because they do not guarantee the cache set to which an allocated block will map, complicating the determination of precise Worst-Case Execution Time (WCET) bounds. This unpredictability hinders static cache analysis in classifying memory accesses as cache hits or misses. Furthermore, modern allocators manage free memory blocks by traversing internal data structures on the heap to find suitable blocks for allocation. These traversals cause loss of cache content information, leading to cache performance unpredictability during allocator execution and resulting in significant overestimations of memory (de)allocation execution times.

The main memory (DRAM) typically provides good average-case response times with minimal fragmentation, which is suitable for most applications. However, it becomes less suitable for safety-critical applications due to its inability to guarantee tight WCET necessary for predictable response times. Consequently, modern

dynamic memory allocators introduce unpredictable cache behaviour, making it difficult to determine precise bounds on AI execution times. Time predictability is a crucial requirement in safety-critical system design [108], as unpredictability in memory allocation and cache performance can lead to significant overestimations of execution times, thereby compromising the reliability and safety of the system.

In safety-critical applications, ensuring predictable response times is essential. The inability of DRAM to provide guaranteed tight WCET makes it less suitable for these applications. The unpredictable cache behaviour introduced by modern dynamic memory allocators further exacerbates the issue, hindering the ability to determine precise execution time bounds for AI systems. This unpredictability in memory allocation and cache performance underscores the need for memory management solutions that can ensure time predictability and maintain the integrity of safety-critical systems [107].

To ensure time predictability, [108] proposed a Reference Counting Memory Management Unit (RCMMU) that facilitates simplified static WCET analysis, thereby reducing the pessimism typically associated with static timing analysis. The RCMMU incorporates a specialised memory arrangement unit that allows object management transactions to occur concurrently with program execution, ensuring no adverse impact on performance. By eliminating the overhead linked to garbage collection operations and negating the need to determine object allocation rates, the RCMMU replaces traditional garbage collectors. Implemented on configurable hardware, such as FPGAs, this time-efficient and statically predictable dynamic memory manager effectively handles dynamic memory without requiring collection pauses, thereby enhancing overall system predictability and performance.

Predictable cache behaviour is essential for accurate timing analysis and deriving tight bounds on a program’s execution time. One approach to addressing the challenge of unpredictability in memory allocation is the Cache-Aware Memory Allocator (CAMA). This innovative constant-time dynamic memory allocator employs cache-aware splitting and coalescing techniques to reduce unpredictability caused by the lack of guaranteed information about the cache set to which an allocated block maps.

CAMA enhances predictability by incorporating a target cache set as an additional parameter during allocation requests. This method relies on the bounded and statically known cache influence of the allocation process, which helps classify memory accesses as either cache hits or misses, thereby enabling precise WCET bounds determination. Furthermore, CAMA improves the management of free memory blocks through its internal data structures on the heap [107]. The algorithm proposed by [107] ensures constant response times and avoids unpredictable cache pollution. By directing allocations to specific cache sets, this approach guarantees that allocated memory is mapped to a predetermined cache set, facilitating precise static cache analysis and thereby improving the overall predictability of memory allocation.

In an effort to enhance program execution times and resource utilisation, [109] introduced a dynamic memory-aware task scheduling strategy for the Spark computing framework, addressing a gap in traditional Spark task scheduling processes. Their dynamic memory-aware task scheduling (DMATS) algorithm dynamically adjusts task concurrency based on feedback information, which helps in managing memory and network I/O as computational resources. This approach aims to op-

timise the utilisation of limited memory resources by adjusting task scheduling in response to current system conditions. These outlined research efforts highlight additional challenges within the realm of reliably and predictably deploying CNN or ANN models to meet the stringent requirements of safety-critical applications, particularly concerning the memory access patterns of neural networks.

2.6 Memory Access issues in Safety-Critical Applications

Memory access issues in safety-critical applications arise from the need for deterministic and reliable performance in environments with stringent timing constraints. Common hardware platforms for neural network deployment often struggle to meet these requirements due to problems such as non-deterministic memory throughput, memory access collisions, and variable latency. These challenges compromise the consistency and predictability essential for ensuring safety and reliability in critical systems.

2.6.1 Hardware Deployment Platforms: Memory Access related Issues for Neural Network Deployment

safety-critical applications demand highly reliable and deterministic performance from hardware platforms for deep learning acceleration such as Common types include Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), Tensor Processing Units (TPUs), and Data Processing Units (DPUs), particularly in terms of memory management. Common hardware platforms for deep learning acceleration face significant challenges in this domain. These platforms encounter issues related to memory throughput, memory access collisions, and memory latency, making them generally less suitable for safety-critical applications.

While considering memory throughput, these platforms, although powerful for general-purpose and high-performance computing, struggle with deterministic memory throughput [55] in safety-critical environments. Their architectures optimise for maximum throughput using massively parallel threads, which can lead to non-deterministic behaviour due to varying access patterns and contention. This variability is particularly problematic as it can result in inconsistent processing times, undermining the reliability essential for safety-critical operations.

Memory access collisions present another significant challenge. The concurrent execution of numerous threads in these hardware platforms leads to frequent memory access collisions [23]. This concurrency results in conflicts and unpredictable memory access times, which compromise the deterministic performance required in safety-critical applications. The lack of fine-grained control over memory access further exacerbates this problem, especially during high-volume batch processing scenarios.

Memory latency issues further complicate the use of common hardware platforms in safety-critical applications. The deep memory hierarchies and high degree of parallelism inherent in these designs introduce significant latency [110] [111], which is non-deterministic and unsuitable for tasks requiring precise timing. The large-scale data movement necessary for deep learning operations contributes to variable

memory latency, which is incompatible with the stringent timing guarantees needed in safety-critical contexts.

Furthermore, these hardware platforms do not prioritise deterministic memory management. The shared resources and parallel processing capabilities, while advantageous for maximising performance, lead to variability in memory access times. This non-deterministic latency, especially under heavy computational loads, renders these platforms unreliable for applications with strict timing constraints.

The complexity of managing memory across distributed systems in some of these platforms also introduces unacceptable variability for safety-critical applications. The concurrent access patterns of various AI workloads in distributed memory systems can lead to significant memory access collisions, posing a risk to deterministic performance. The complex memory systems in these platforms, designed to handle large-scale AI workloads, face challenges in providing consistent and predictable performance, which is a critical requirement for safety-critical tasks.

While common hardware platforms for neural network acceleration are highly efficient and powerful for general AI workloads, their design limitations in terms of memory throughput, memory access collisions, and memory latency present substantial barriers to their application in safety-critical domain. The non-deterministic behaviour arising from their architectural optimisations and concurrent processing capabilities undermines the reliability and precise timing essential in these environments.

2.6.2 Existing Hardware Frameworks for Inference Acceleration: Unsuitability for Safety-Critical Applications

Common hardware and inference acceleration frameworks such as Vitis AI, VTA, FINN, and HLS4ML are designed to accelerate neural network inference, offering high performance and efficiency. However, when it comes to safety-critical applications, these platforms face significant obstacles in ensuring reliable and deterministic performance. These challenges are particularly pronounced in the domain of memory management, which is critical for the predictability and stability required in safety-critical systems. These frameworks and tensor accelerators exhibit issues related to memory throughput, memory access collisions, and memory latency, making them generally less suitable for safety-critical applications.

- **Vitis AI** relies on FPGA architecture, which, while highly flexible and efficient, can suffer from variable memory access times due to resource sharing and dynamic reconfiguration. This variability impacts its suitability for safety-critical tasks, where consistent memory access is crucial. The shared memory architecture in Vitis AI can lead to access collisions when multiple processes compete for the same resources, resulting in non-deterministic behaviour that undermines the reliability required in safety-critical environments. Additionally, Vitis AI's FPGA architecture can introduce memory latency variability due to dynamic reconfiguration and resource contention, further compromising real-time performance.
- **VTA**, designed for deep learning research, exhibits high memory throughput variability. Its use of a programmable accelerator, while offering flexibility,

exposes it to inconsistent memory access times, making it unreliable for applications that require stringent timing guarantees. The programmable nature of Apache TVM-VTA increases the likelihood of memory access collisions, particularly when multiple tasks access shared memory concurrently. This can lead to unpredictable delays and compromises the deterministic behaviour necessary for safety-critical applications. Furthermore, VTA’s memory latency can be unpredictable due to its programmable nature, complicating timing analysis and making it less suitable for applications requiring precise timing.

- **FINN**, an FPGA-based framework, offers customisability but suffers from inconsistent memory throughput due to variability in FPGA fabric utilisation and reconfiguration delays. This inconsistency can lead to unpredictable performance, which is detrimental in safety-critical scenarios where reliable and consistent memory access is essential. FINN’s FPGA-based approach can also suffer from memory access collisions, especially when multiple logic blocks attempt to access shared memory resources simultaneously. These collisions can cause delays and non-deterministic behaviour, further reducing its suitability for safety-critical tasks. Additionally, FINN’s design can experience memory latency issues due to reconfiguration times and the inherent variability in FPGA resource allocation and access patterns, affecting the consistent performance needed in safety-critical applications.
- **HLS4ML** enables the deployment of machine learning models on FPGAs, but it can encounter variable memory throughput due to the high-level synthesis process and FPGA resource allocation. This variability can lead to inconsistent performance, which is problematic for safety-critical applications that demand reliable and deterministic memory access. The FPGA implementations generated by HLS4ML may experience memory access collisions when multiple synthesised modules attempt to access shared memory resources, leading to unpredictable delays and reducing reliability. HLS4ML can also suffer from memory latency issues, where the synthesised hardware design may not provide consistent and predictable memory access times, affecting real-time performance in safety-critical applications.

2.6.3 Memory Interference in Neural Network Deployment

Memory interference significantly impacts the stability and predictability of neural network deployment in hardware. This interference can arise from various sources and manifests in different ways, affecting the performance of tensor accelerators and reliability of neural network memory accesses. Key aspects of memory interference include:

- **Shared Resource Contention:** When multiple neural networks or applications concurrently access shared memory resources, contention arises. This contention leads to delays and unpredictable access times, which can degrade the performance of neural network deployments on accelerators.
- **Cache Coherence Overhead:** Maintaining cache coherence in multicore and many-core systems introduces overheads that can interfere with memory access patterns. The synchronisation required to ensure consistency across caches can

cause delays and reduce the effective bandwidth available to neural network operations.

- **Memory Bandwidth Saturation:** Accelerators often share memory bandwidth among multiple processes. When the cumulative memory bandwidth demand exceeds the available capacity, it results in saturation, leading to increased latency and reduced throughput for neural network computations.
- **Non-uniform Memory Access (NUMA) Effects:** In systems with non-uniform memory access architectures, the distance between memory and processing units can vary. Accessing remote or off-chip memory can incur higher latency compared to local memory, causing variability in memory access times and affecting neural network performance.
- **Interference from Background Processes:** Background processes running on the same hardware platform/system as the neural network accelerator can cause interference by competing for memory and other system resources. This competition can lead to fluctuations in memory access times and system unpredictability.
- **Dynamic Memory Allocation:** The dynamic nature of memory allocation and deallocation during neural network execution can lead to fragmentation and inefficient use of memory resources. This fragmentation can cause delays in memory access and impact the predictability of neural network performance.
- **Priority Inversion:** In systems where different tasks have varying priorities, lower-priority tasks may hold memory resources needed by higher-priority neural network tasks, causing delays and unpredictability in the execution of high-priority tasks.
- **Memory Access Patterns:** The specific access patterns of neural network workloads, such as irregular or bursty accesses, can lead to contention and interference when combined with other workloads with different access characteristics. This interference can cause unpredictable variations in memory access latency and throughput.

These factors collectively contribute to the complexity of managing memory resources in neural network deployments on accelerators and require optimisations to make existing accelerator better suited for neural network application in the safety-critical domain.

2.7 Memory Access Management and Optimisation: A Review

Ongoing research efforts aim to address the challenges and limitations outlined in previous section, focusing on improving neural network models, redesigning accelerators, and adopting various optimisation strategies to enable the efficient deployment of neural networks. These efforts are particularly crucial in safety-critical applications, where reliability, predictability, and safety are paramount.

2.7.1 Researches on Neural Networks towards Reliability in Safety-Critical Applications

To facilitate the use of neural networks in safety-critical systems while addressing concerns related to safety, security, and predictability, [88] proposed an integrated software architecture. This architecture combines multiple technologies to perform deep learning inference while ensuring inherent safety, security, and predictability. The key components of this architecture include:

- **Hypervisors:** Hypervisors are used for comprehensive runtime monitoring, providing isolation of components based on their criticality and security needs. This isolation helps protect essential components from failures and cyber threats by keeping them in distinct execution domains.
- **Redundancy and Diversity:** These strategies enhance the robustness of deep learning components, mitigating the risks associated with faulty neural network outputs in critical scenarios. By employing diverse methods and redundant systems, the architecture increases fault tolerance.
- **Predictive Fault Detection and Recovery:** The architecture incorporates predictive fault detection mechanisms to anticipate potential faults through simulations using digital-twin technology. In case of detected faults or unreliable neural network responses, a recovery mechanism is triggered, switching control from DNN-based controllers to simpler, safer controllers.
- **Predictable DNN Inference Engine:** A key innovation proposed by [88] is the development of a predictable DNN inference engine that reduces interference among multiple concurrent DNNs operating at different execution rates on the same platform. This engine ensures that real-time performance requirements are met, a critical aspect in safety-critical applications.

Further exploring software applications in safety-critical domains, [112] identified that existing architecture patterns often fail to meet non-functional requirements such as safety and modifiability, particularly in the context of automated driving. To address these limitations, [112] proposed a new generic architecture pattern targeting ISO 26262 safety standard compliance, though their work did not specifically focus on DNNs.

In the realm of programming languages and architecture for time predictability, [1] tackled the limitations of the C programming language in providing seamless design and time predictability for concurrent systems. They introduced PRET-C, an extension of C for deterministic and reactive concurrent programming, along with the ARPRET architecture, a precision-timed machine designed to ensure time-predictable execution through customisable soft-core processors.

Regarding the timing predictability of neural networks, several researchers have analysed GPU-based accelerators to propose solutions. For instance, [113] developed a system to optimise DNN workload execution on GPUs in real-time multi-tasking environments. Their approach balances real-time correctness and throughput by using a governor for intelligent data fusion and a scheduling framework that incorporates deadline-aware scheduling and CUDA stream concurrency. This system

maximises performance and resource utilisation, particularly in video streaming applications.

Similarly, [114] proposed methods to overcome the limitations of CNN frameworks on embedded hardware. They introduced techniques for processing multiple video streams efficiently through DNN decomposition, parallelism, pipelining, and image merging. These techniques enhance the throughput of multi-camera detection tasks in automated driving systems without incurring significant latency or accuracy loss, thereby contributing to the real-time capabilities essential in safety-critical applications.

These studies highlight the ongoing efforts to enhance the reliability, predictability, and safety of neural networks in critical applications. The approaches range from software architectural improvements to specialised programming languages and hardware optimisations, each addressing different aspects of the complex challenge of deploying neural networks in safety-critical environments.

2.7.2 Researches on Hardware Tensor Accelerators for improving Neural Networks Deployment

Tensor accelerators and deep learning hardware accelerators have received significant attention in recent years due to the increasing demand for efficient computation of DNN. Several research efforts have focused on designing specialised architectures and accelerators to accelerate tensor operations and improve deep learning performance.

While common hardware platform such as NVIDIA GPUs, Google TPUs, FPGAs, Intel Nervana NNP, Habana Labs Gaudi, and Facebook’s Zion offer significant computational power for deep learning and other high-performance applications, they face substantial challenges in meeting the stringent requirements of safety-critical applications. As established in Section 2.6, issues related to memory throughput, memory access collisions, memory issues, and memory latency undermine their reliability and determinism. Furthermore, while Vitis AI, Apache TVM-VTA, FINN, and HLS4ML offer significant capabilities for deep learning and machine learning applications, the inherent memory access issues including memory throughput, memory access collisions, and memory latency make them less suitable for safety-critical applications that require highly reliable and deterministic performance.

One prominent work in this area is the Tensor Processing Unit (TPU) developed by [63]. The TPU is a custom-designed ASIC that provides high-performance and energy-efficient execution of tensor computations in datacentre environments. It leverages a systolic array architecture and a dedicated memory hierarchy to deliver high throughput for deep learning workloads. Another notable contribution is the Eyeriss accelerator proposed by [115]. Eyeriss is designed specifically for CNNs and employs a spatial architecture that exploits data reuse and locality. It incorporates specialised dataflow and memory structures to optimise the performance of convolutional operations.

Several other tensor accelerators and deep learning hardware accelerators have been proposed, each with their unique architectural features and optimisations [116] [117] [118]. These works have demonstrated the importance of specialised accelerators in achieving the efficient execution of deep learning algorithms.

2.7.3 Researches on Neural Network Memory Interference

Addressing memory interference requires careful consideration of system architecture, resource allocation strategies, and workload characteristics to ensure stable and predictable performance. Previous research has proposed various solutions to address interference within neural networks. [119] identified varying degrees of interference within the network, caused by the lack of segregation of input space for CNNs during training. They proposed an interference-less neural network training (ILNNT) method by grouping attributes of mutual benefit connected to the same hidden neurons while separating interfering attributes in different batches to prevent them from sharing the same hidden neurons. This method reduces interference among input attributes.

In recent times, the architecture of specialised acceleration platforms has achieved significant success in processor architecture and trending developments in computer architecture. However, the memory access performance remains relatively poor, limiting the overall performance improvement of hardware accelerators. Additionally, research on memory controllers for hardware acceleration platforms has been limited and scarce. Common hardware acceleration platform designs often neglect layer diversity in terms of computation and communication behaviour. Consequently, on-chip memory resources are underutilised for memory-bounded layers, leading to suboptimal performance [120].

[120] proposed a layer-conscious memory management framework for FPGA-based DNN hardware accelerators that exploits layer diversity and the disjoint lifespan information of memory buffers to utilise on-chip memory efficiently. This approach aims to improve the performance of memory-bounded layers and addresses issues of suboptimal performance caused by the underutilisation of memory resources through on-chip memory allocation, thereby enhancing the performance of DNNs.

[121], the most closely related work to this paper, proposed a specialised memory controller for neural network accelerators, known as the dynamic random access memory (DRAM) memory controller (NNAMC). This controller monitors the memory access stream of an accelerator and dynamically transfers it to the optimal address mapping scheme based on the memory access characteristics, thereby enhancing memory access performance. Their solution analyses the type of data stream accessed by the accelerator using hardware and designs the address mapping for different banks through a bank partitioning model (BPM). Their approach achieves significantly lower access latency for the hardware accelerator compared to competing address mapping schemes.

In their work, [121] utilised the Parameter Reference Table (PRT) to store address streams, network parameters, and system operation results in internal entry tables. When the accelerator accesses the memory system, the image length and window size in these entry tables are updated in response to the storage requests from the accelerator. Their analysis of the address sequence accessing a segment instance identified row conflicts, which can severely impact the memory system by hindering other operations in the same bank, leading to variable memory access operations. Their proposed NNAMC system exploits the variability in memory access to optimise DRAM performance and reduce row conflicts.

[121]’s solution specifically focuses on CNN-based accelerators for image processing. They applied the Address and Pixel Transaction (APT) process on the accelerator to store image data in DRAM and read pixel data using a CNN-like

computing operation, utilising address mapping to map image pixels to the memory system. While their work is restricted to image pixels with CNN, our research addresses a broader range of neural networks, including ANNs, CNNs, and beyond. Although their method reduces memory access conflicts, it does not completely mitigate contention problems and does not address the predictability of memory accesses.

Neural network execution is highly data-intensive, requiring frequent memory accesses and significant computing power. To meet these demands, contemporary multi-processor systems-on-chip are increasingly reliant on large, high-speed external memories, such as DDR2 SDRAMs, shared between multiple intellectual property (IP) cores to balance memory requirements with cost efficiency [110]. However, DRAMs exhibit highly variable access times influenced by previous requests, making it difficult to meet hard real-time requirements due to unpredictable latencies and limited ability to harness useful bandwidth during design [110]. The access patterns within the “3-D” structure of DRAM chips significantly affect system bandwidth and latency [122]. Unordered memory accesses can lead to contention and an increase in task’s WCET, especially when multiple CPU cores and DMA peripherals run concurrently. For instance, in an eight-core system, tasks that spend only ten percent of their time fetching data from memory may experience a 300 percent increase in WCET due to memory interference [122].

[122] proposed a method to compute the upper bounds on task delay due to memory contention in multicore systems, using this computed delay to schedule tasks based on assigned time slots for each core. They analysed maximum memory traffic and cache behaviour to generate delay bounds, which guide scheduling to minimise memory contention. Another solution to enhance memory system performance is the reordering of memory operations. [111] proposed memory access scheduling strategies that maximise memory bandwidth by exploiting the locality within the 3-D structure of DRAM, improving performance through conservative and aggressive reordering of memory references to optimise bandwidth utilisation. [110] designed a predictable memory controller using specialised hardware integrated into the network interface of a network-on-chip. This controller identifies memory access groups with known efficiency and latency, dynamically scheduling these groups to allocate bandwidth and ensure a maximum latency bound, providing predictable memory performance. However, this hardware approach is tedious, costly, and unsuitable for AI applications.

In another study, [123] proposed a software-based memory throttling controller to manage memory interference in shared memory multicore systems. This controller uses a dynamic throttling strategy to protect critical tasks by partitioning them and periodically restricting memory read/write accesses among cores, monitored by an OS scheduler. Although this approach satisfies the schedulability of critical tasks, it is complex and applies only to multicore memory access without considering the memory access dynamics of hardware accelerators or neural network acceleration.

2.7.4 Researches on Memory Access Optimisations

Neural networks have become integral in various domains, including image classification, object detection, and natural language processing, due to their ability to handle complex data and deliver high accuracy. These networks are composed of

multiple layers, each executing specific operations such as convolution, pooling, and fully connected layers. The performance of neural networks is heavily influenced by memory access patterns. Inefficient memory access can result in significant delays, reducing the overall throughput and execution speed of the network. Addressing memory access issues is therefore crucial for optimising the performance of neural networks and enabling their effective deployment in real-world applications.

Previous research has explored various strategies to optimise memory access in neural networks, addressing both hardware-level optimisations and software-based scheduling techniques. One significant contribution in this area is the development of the Prefetch-Aware DRAM Controller (PADC) [124], designed to balance the prioritisation between demands and prefetches in memory controllers. Traditional controllers either treat prefetches the same as demand requests or always prioritise demands, leading to suboptimal performance due to bandwidth consumption by useless prefetches or delays in useful prefetches. The PADC addresses this issue by adaptively prioritising between demands and prefetches based on prefetch accuracy, and by dropping useless prefetches to free up memory resources, resulting in significantly improved performance over rigid prefetch policies [124].

Another study focused on improving bank-level parallelism in multiple memory banks to reduce contention involved redesigning DRAM and Non-volatile Memory (NVM) controllers [125]. This research introduced a hardware-assisted checkpointing mechanism to mitigate memory bank conflicts between applications. Traditional checkpointing mechanisms in hybrid memory systems often ignore the impact of memory contention, leading to performance degradation. The proposed solution schedules memory requests to utilise idle time slots in different memory banks and promotes the priority of application memory requests to reduce access latencies when conflicts are unavoidable. This redesign allows data to be directly transferred from working memory to checkpoints in NVM without CPU intervention, significantly improving performance and reducing latency [125].

While various strategies have been proposed to optimise memory access in multi-core and shared memory systems, these approaches often fail to address the unique challenges posed by hardware accelerators and neural network workloads. This dissertation proposes an augmented platform that ensures system predictability through timing guarantees provided by a time-triggered architecture. It also leverages cache memory to reduce the burden on main memory, utilising idle phases of DRAM to improve speed and efficiency, thus addressing the growing computational and memory demands of safety-critical systems equipped with AI functionality.

Furthermore, while these approaches focus on optimising allocators to improve specific program predictability, execution times, and resource utilisation, they do not explicitly address the dynamics of neural network weight and bias loading or the computational behaviour of memory accesses in the context of AI accelerators. These aspects are critical as they impact the performance of AI applications, especially in safety-critical systems.

In addition, the proposed memory optimisation techniques in earlier research have not been specifically tailored to ensure the suitability of hardware accelerators for safety-critical applications. This dissertation proposes techniques such as utilising time-triggered architecture for memory accesses, implementing time-based schedules to prevent memory access collisions, employing predictive memory management by prefetching data ahead of requests and using deterministic data place-

ment by caching frequently accessed data.

This dissertation strives for predictability in AI applications in safety-critical systems by utilising time-triggered architectures. Additionally, it aims to improve execution times and reduce DRAM resource utilisation through a memory access reordering approach and advanced memory optimisation techniques. By addressing the specific challenges posed by neural network computations and memory access patterns, our method provides a comprehensive solution for enhancing the performance and predictability of neural networks across various applications.

2.8 Leveraging Time-Triggered Approaches for Predictability

Safety-critical systems, such as those used in healthcare, automotive, avionics, and industrial control—require stringent guarantees of determinism, fault tolerance, and timing predictability. These systems must operate reliably and predictably to prevent catastrophic failures that could endanger lives or cause significant property damage. To meet these demanding requirements, time-triggered architectures have shown considerable potential and have been widely explored and utilised by researchers.

Time-triggered architectures offer deterministic execution and temporal predictability [126] by scheduling tasks to be performed within predefined time windows. This approach ensures that operations occur at precisely specified intervals, thereby reducing variability and increasing reliability in systems where timely responses are critical. Several applications have demonstrated that leveraging time-triggered strategies can effectively achieve the desired levels of predictability and reliability [27]. By adhering to strict time schedules, these architectures contribute to meeting the rigorous demands of safety-critical environments.

Time-triggered approaches involve scheduling tasks based on a global clock, with each task assigned a specific time slot for execution. This contrasts with event-triggered systems, where tasks are executed in response to events, leading to potential timing unpredictability. The deterministic nature of time-triggered systems is particularly advantageous for real-time and safety-critical applications, where ensuring tasks are completed within their allotted time slots is essential.

Time-triggered approaches can provide the necessary predictability and reliability in the context of neural network deployment on FPGAs for safety-critical applications. By scheduling the execution of neural network operations based on a global clock, it is possible to ensure that tasks are completed within their assigned time slots, reducing the risk of timing unpredictability. This method also facilitates the management of memory access patterns, ensuring that data is fetched and processed predictably.

Implementing time-triggered strategies in FPGA-based neural network deployment requires designing hardware architectures that support time-triggered execution. This involves creating mechanisms to manage the timing of data transfers, computation, and memory access, ensuring that all operations adhere to the predefined schedule. Such designs significantly enhance the predictability and reliability of neural network deployments in safety-critical applications, addressing the challenges of temporal unpredictability and resource constraints.

Research in this field has focused on various aspects, including scheduling algorithms, communication protocols, and fault tolerance mechanisms. For example, [127] introduced a time-triggered processor architecture for real-time systems, enabling the execution of time-triggered tasks on multicore processors. Their architecture features a time-triggered instruction pipeline, where instructions are fetched and executed according to a predetermined schedule, ensuring strict temporal isolation and guaranteeing bounded execution times for critical tasks [127].

Similarly, [128] proposed a flexible time-triggered architecture for mixed-criticality systems. This architecture supports tasks of varying criticality levels and provides temporal partitioning to ensure isolation between tasks. It also incorporates dynamic scheduling techniques to adapt to changing workload requirements while maintaining temporal predictability [128]. Furthermore, Chen et al. (2014) introduced a time-triggered real-time system architecture designed to provide predictable execution of tasks with bounded delays [117].

These studies highlight the importance of time-triggered architectures in achieving reliable and predictable execution for time-critical applications. However, their application to neural network deployment within the safety-critical domain remains underexplored. The insights gained from these architectural studies are instrumental in designing the proposed time-triggered caching in this dissertation. By leveraging time-triggered execution principles and integrating them with deep learning hardware acceleration, this work aims to achieve efficient and deterministic execution of deep learning workloads in safety-critical environments.

2.9 Contribution to Memory Access Predictability and Optimisation

The increasing complexity of DNN structures exacerbates challenges in on-chip memory allocation, often impacting memory utilisation efficiency. Efficient memory performance is intricately linked to the access patterns of applications. Since arithmetic operations in accelerators demand high memory bandwidth, memory access contention can occur, severely disrupting memory systems and affecting other concurrent operations, leading to variable memory access patterns [121].

A central focus of this dissertation is the computation, tracing, and analysis of memory access patterns. As noted by [121], analysing these patterns is complex and poses significant challenges. Despite various advancements in the field, as discussed in Section 2.6, there has been limited attention to system-level and architectural aspects of memory access in neural networks for safety-critical applications. Issues related to predictability, inference speed, and resource requirements, particularly in conjunction with critical hardware components, require further exploration.

Furthermore, the proposed memory optimisation techniques in previous research works have not been specifically tailored to enhance the suitability of neural network hardware-based accelerators for safety-critical applications. This dissertation proposes techniques such as utilising time-triggered architecture for memory access, implementing time-based schedules to prevent memory access collisions, employing predictive memory management through prefetching, and employing deterministic data placement through caching. These techniques aim to improve memory access performance and significantly reduce access latency, thereby enhancing the reliabil-

ity of hardware accelerators in safety-critical environments.

By addressing these challenges to enhance the suitability of existing hardware accelerators for safety-critical applications, this dissertation presents the following techniques:

- **Time-Triggered Memory Access:** Utilising a time-triggered architecture for memory access can enhance determinism. By scheduling memory access operations based on fixed time intervals, variability can be reduced, and consistent performance can be ensured.
- **Memory Access Time-Based Schedules:** Developing schedules that allocate specific time slots for memory operations can mitigate collisions and latency. This approach ensures that memory operations receive timely access to memory resources.
- **Predictive Memory Management with Prefetching Mechanisms:** Memory prefetching, established as an effective method in previous research, can further reduce latency and improve on-chip memory utilisation. By anticipating memory access patterns and reordering execution by prefetching data, delays can be minimised. Loading necessary data into faster, local memory preemptively helps to decrease access time.
- **Data Caching with Deterministic Data Placement:** Organising data in memory to ensure predictable access times can enhance overall system performance. Placing frequently accessed data in low-latency memory regions (cache) reduces access times to the main memory and improves determinism.

Implementing the advanced memory optimisation techniques discussed—such as time-triggered systems, time-based memory access schedules, caching and predictive memory management—can improve the suitability of accelerators for safety-critical applications. These optimisations ensure more predictable and reliable performance, which is crucial for applications where safety and precision are essential.

This dissertation explores hardware- and software-based techniques to effectively address the outlined challenges and provide reliable, cost-effective solutions. As highlighted by [32], software-based techniques are particularly suitable for profiling individual VTA module performance or evaluating the overall performance profile of model data processing within the accelerator during simulation. These techniques allow for the examination of optimisation strategies before their incorporation into the hardware design. However, during the design phase, when accurate performance data of the model running on the VTA is required—especially for accurately populating the access time slices in the schedule and accounting for data transfer latencies between on-chip and off-chip memory, which are not adequately modelled by software simulations—hardware benchmarking becomes essential.

Accordingly, this dissertation begins with hardware-based traces to capture accurate performance metrics, as detailed in Chapter 3 and complements this with software-based simulations in Chapter 4 for modelling and evaluating the proposed optimisation strategies. During the optimisation phases, a software simulator is utilised to alleviate the costs associated with hardware synthesis [129] until the optimisation strategies are fully conceived, developed and evaluated. Once these strategies are validated, the accelerator design is mapped to the target FPGA, where

actual performance metrics can be gathered for further hardware evaluation. This process of hardware-synthesised design evaluation continues iteratively until the final design is achieved, meeting the optimised time-triggered system’s expected performance targets, such as predictability through a suitable schedule, improved memory throughput free of contention, and reduced inference time.

To the best of the author’s knowledge, no comprehensive architecture has yet been designed that integrates time-triggered strategies with memory access optimisation techniques to improve inference speed while ensuring predictability, reliability, time guarantees, and safety. This dissertation seeks to address this gap by analysing memory access behaviour and patterns in deep learning accelerators and developing advanced memory access controllers for existing hardware tensor accelerators. By focusing on these critical aspects, the research aims to enhance the predictability and reliability of neural network deployments in safety-critical applications.

Chapter 3

Time-Triggered Caching

In existing neural network accelerators, such as the Versatile Tensor Accelerator (VTA), system performance is highly dependent on the efficiency of memory access operations. Moreover, the system requires timing guarantees to ensure suitability for safety-critical applications. This chapter introduces a time-based system model, its co-design with VTA architecture to ensure the predictability of memory accesses while applying advanced memory management techniques aimed at optimising memory footprints during neural network access to Dynamic Random Access Memory (DRAM), also referred to as *main memory* or simply *memory* throughout this chapter.

3.1 Time-Triggered Caching System Model

As depicted in Figure 3.1, the system model illustrates the time-triggered memory access protocol, integrating time-triggered components with the VTA to establish a complete hardware-software stack interface. This time-triggered memory cache framework ensures predictable and deterministic inference execution on the VTA.

In developing high-performance systems, particularly those utilising multicore interfaces with FPGAs, specific software methodologies are required to enhance system performance, maintain resource coherency, and efficiently manage memory. This section details the proposed system integration, incorporating technical designs and strategies to address challenges such as predictable accesses, cache coherency, memory isolation, buffer management, and neural network optimisation. These are essential for improving computational performance on FPGA-based machine learning systems. The following software components are implemented to ensure safety conformity and consistency in performance.

- **Cache Coherency Management in Multicore Platforms:** In multicore platforms such as the Zynq UltraScale MPSoC, cache coherency is crucial for ensuring consistent data access across processing units. The system model incorporates a cache coherence management framework that maintains consistent memory states between the Application Processing Unit (APU) and external interfaces [130].

A custom cache coherency mechanism ensures efficient data transfer between the VTA and CPU. This mechanism coordinates memory access and ensures that all components within the system have coherent and synchronised ac-

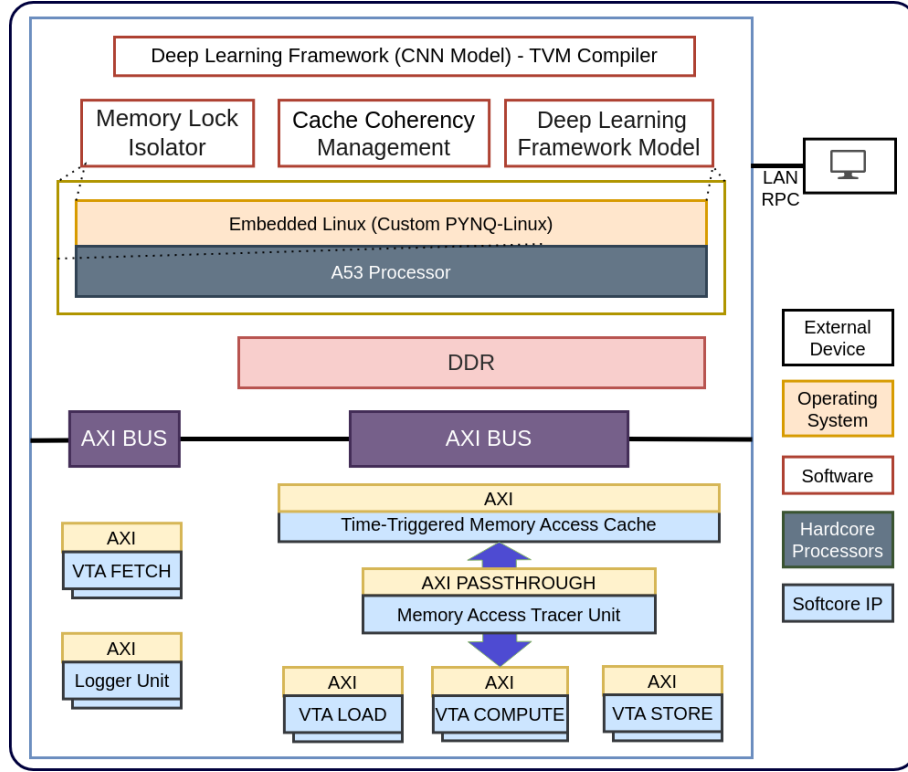


Figure 3.1: System Architecture of Time-Triggered Memory Cache

cess to shared memory regions, thereby improving performance during neural network inference operations.

- **Memory Lock Isolation:** Simultaneous memory requests from multiple threads can lead to resource contention in systems with high concurrency, such as those running PYNQ Linux. A memory lock isolation technique is introduced to control memory access and prevent race conditions [131]. This technique locks memory buffers to specific regions [132], ensuring that critical data such as those from neural networks remain secure and available for active processes. Additionally, memory allocation strategies prioritise access based on system demands, reducing contention and ensuring efficient memory utilisation across concurrent threads.

- **Buffer Allocation:** Effective buffer management is vital to optimising performance for machine learning tasks on FPGA platforms. Memory buffers are allocated based on the requirements of the neural network model and the architecture of the VTA [133].

A unified memory buffer allocation strategy is utilised with a known entry point to enhance determinism by controlling external operations accessing the memory from contention with the VTA memory operations. This approach ensures that buffer slots are efficiently managed, allowing for predictable memory access patterns and reliable execution during inference on FPGA platforms.

These software components interface directly with the TVM compiler to facilitate model deployment. Upon launching the pre-trained model on the host, the cache coherency software ensures coherent data access between the VTA and memory. These, the memory allocator manages buffer assignments to prevent address

conflicts during execution. These components ensure the system’s robustness, handling tasks like cache management for multicore hardware and memory isolation, enabling efficient and predictable operation.

The VTA modules, discussed in prior chapters, have been restructured to interface with the time-triggered system through its AXI interconnect. The time-triggered controller, a central component of the time-triggered cache, provides the necessary infrastructure to enable deterministic, time-triggered memory access, ensuring predictable execution. The time-triggered mechanism also includes the schedule dispatcher and the memory access control strategies that access the allocated buffers to the VTA, ensuring safe and controlled memory interactions. The AXI bus interface enables structured and timely memory access.

The time-triggered memory cache is deployed on a Xilinx Zynq UltraScale+ MPSoC FPGA platform, such as the TE0802, following a precomputed schedule generated through an analysis of memory access patterns. The dispatcher manages this schedule, which orchestrates all memory operations while mitigating contention and reducing latency variations commonly encountered in conventional VTA memory operations.

The precomputed schedule, derived from the analysis of memory access traces, governs the timing of all memory operations, ensuring that data is retrieved ahead of VTA requests. The trace module within the memory access tracing unit logs all memory operations accessible to the processor via a memory-mapped interface. This mechanism facilitates efficient data retrieval and analysis for memory access scheduling, allowing all memory operations to be time-triggered, thereby guaranteeing predictable performance throughout the inference process.

This dissertation introduces a novel approach to achieving predictable and deterministic memory access for neural network inference, referred to as the Time-Triggered Memory Access Versatile Tensor Accelerator (TTmaVTA). By integrating time-triggered memory access mechanisms into the VTA architecture, the TTmaVTA framework addresses the limitations of traditional accelerators in managing memory bandwidth, resource contention, memory access inconsistency and latency. The TTmaVTA improvements make it a suitable solution for safety-critical AI applications, ensuring predictable inference performance.

3.2 Time-Triggered Memory Access of Tensor Accelerators

This dissertation focuses on leveraging the Versatile Tensor Accelerator (VTA), a widely recognised open-source tensor accelerator, to conduct an in-depth analysis of memory accesses in neural networks to manage and control memory accesses. The findings from this analysis are intended to optimise memory footprints, ultimately contributing to predictable performance and efficiency of tensor accelerators. The Time-Triggered Memory Access process system, as depicted in Figure 3.2, comprises four distinct modules categorised based on their functions at the system’s design stage and implementation. The system is broadly divided into two main phases:

- **Phase 1: Memory Access Tracing, Analysis, and Scheduling:** The first two modules Figure 3.2 are utilised in this phase, with the second conducted

as an offline operation. These modules track and record the VTA's memory access patterns during runtime to aid in generating a suitable schedule. The traced data are analysed offline using network flow graphs and custom software tools. This analysis facilitates understanding memory access patterns, the creation of an appropriate schedule for time-triggered memory access control, and identifying opportunities for optimisation. These functionalities are essential for the efficient management of memory usage and behaviour.

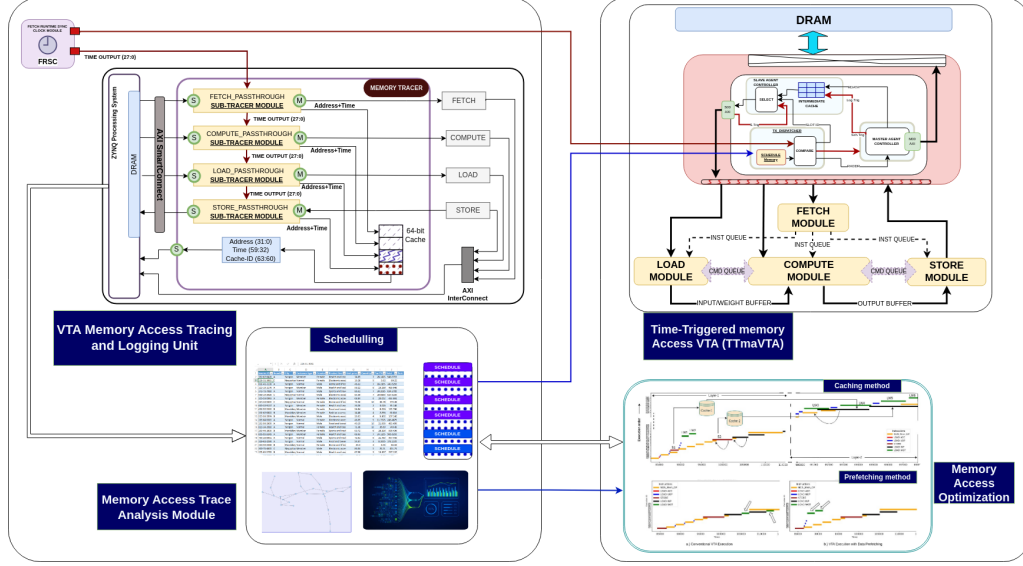


Figure 3.2: Time-Triggered Memory Access Architecture

- Phase 2: Time-triggered Memory Access Control and Optimisation:**
 This phase consists of modules responsible for managing time-based control of memory access according to the schedules generated in Phase 1. These modules ensure that memory operations and accesses to the main memory are synchronised and executed based on predefined timing constraints, optimising performance and maintaining system integrity. The final module in this phase focuses on implementing optimisation strategies derived from the analysed data, such as prefetching data during idle periods while waiting for other memory accesses and caching frequently used data to minimise main memory access. These strategies aim to enhance memory throughput, improve performance, and accelerate memory access operations.

The division into these two phases helps organise the system's functionality and focus on specific memory management and control aspects. The components/modules that are functional in Phase 1 include Memory Access Tracer and Logger Unit (MATL) performed at runtime and the Memory Access Trace Analyser. The main contributions of this chapter are summarised as follows:

- Conducting memory access measurements using a hardware-based technique with custom IPs. The measured trace data is logged into a cache and accessed by the Processing System (PS) via a memory-mapped interface utilising MMIO. The data is retrieved by a Linux-based application running on the PS to examine and analyse VTA memory access comprehensively.

- Memory access pattern analysis of the logged parameters and a schedule generation software tool utilising scheduling algorithms are integrated into the offline stage. System performance is improved through the utilisation of the generated schedule, which is based on the identified and analysed memory access streams. This optimisation effectively mitigates contentions caused by memory access collisions due to concurrent memory operations, ensuring more predictable memory access.
- The TTmaVTA design is based on an FPGA hardware development system, incorporating a memory access controller (TTmaC) dedicated explicitly to neural network accelerators like the VTA. The controller is implemented in Very High-Speed Integrated Circuit Hardware Description Language (VHDL), minimising reliance on software operations.

3.3 Tracing and Logging of Memory Access Transactions

Multicore architectures are increasingly popular for deploying neural networks in embedded platforms due to their ability to improve throughput and reduce power consumption across multiple application workloads. These architectures allow cores to share hardware resources, such as caches and memory controllers, which introduces non-deterministic interactions between threads and processes in the memory system. This leads to unpredictable memory access patterns, significantly impacting system predictability and performance [134].

To address these challenges, many dependable real-time architectures have adopted a time-triggered paradigm to capitalise on predictable timing behaviour [127]. The time-triggered approach requires a priori knowledge, including the type of memory accesses, execution times and minimal inter-arrival times for all tasks—a difficult and labour-intensive process [135], but essential for creating a schedule to aid in time-triggered system implementation.

A primary component of a time-triggered memory system is the schedule, which requires prior knowledge of the system, such as the VTA instructions and memory access patterns, including addresses and sequences of operation execution. Having prior knowledge necessitates an in-depth exploration of the VTA and its memory operations, encompassing data acquisition techniques, sorting, filtering, and generating resultant files for further analysis. After analysis, the schedule is generated which must account for evolving circumstances and changing requirements, necessitating revisions with every software or hardware change [127]. This constant need for adaptation complicates the time-triggered process, making it more challenging to implement and requiring a deep understanding of event-triggered resources of the VTA, to adapt the schedule effectively.

Furthermore, the characteristics of CNN convolution operations, as discussed in Section 2.1, highlight common issues during accelerator memory access, such as row conflicts, which contribute to delays in memory access. These conflicts occur due to the series of two-dimensional accesses to memory during convolution computations performed by the processing elements, leading to repeated precharging and reactivating of memory rows, which in turn increases access latency [121].

In addition to delays in memory accesses, several factors influence memory access patterns, including the characteristics of CNN computations, such as operation sequences, two-dimensional access patterns, stride lengths, input feature maps, varying weight sizes, and random memory accesses [121]. For scheduling memory access, additional critical factors such as the specific memory address and the timing of access must be considered to evaluate access patterns and design an optimal schedule for memory accesses. These factors are particularly important for identifying the best scenarios from the traced data to optimise memory utilisation and throughput.

Understanding memory access in VTAs is essential for tracing, logging, and analysing memory access patterns, particularly for computationally intensive operations such as neural network computations. Furthermore, interference from other applications, such as the PS and other IP cores, may affect memory access, leading to resource contention and unpredictable memory access patterns. This dissertation presents two memory tracing approaches: a hardware-based tracer and a software-based tracer. While both methods effectively trace memory access, it is essential to recognise that memory access patterns differ between neural networks [136] [137]. Consequently, the memory access traces for one neural network will vary from another. For example, memory access traces for ResNet differ from those for Yolo. The design considerations for both hardware and software tracing approaches, along with their trade-offs, are discussed. Software-based full memory access tracing is often flexible but provides only estimates generated through software simulations [32]. In contrast, the hardware-based tracing approach, while cumbersome [138] and costly [134], has the advantage of capturing actual performance values [32] by tracing memory access patterns on a VTA-synthesised accelerator running on an FPGA.

This chapter presents a hardware-based method for collecting full memory access traces, offering the potential to reduce overhead. However, due to the massive data volumes generated by the neural network during execution, this approach also presents challenges, such as resource constraints, especially on FPGA-based platforms. Therefore, this chapter explicitly traces the memory access patterns for the optimised 2D Convolutional Neural Networks (Conv2D) model.

The contributions to memory access tracing and pattern analysis presented in this chapter include:

- A detailed study characterising hardware-based memory access trace at run-time across various layers of specific models.
- A demonstration of FPGA-based acquisition and logging of memory access traces in VTA architecture with Conv2D workloads.
- Analysis of memory behaviours across multiple runs to compute minimum and maximum latencies, providing worst-case scenarios.
- Examination and evaluation of memory access data using network flow graphs and software analytical tools to generate insights for system optimisation

A complete memory access operation is a round-trip process comprising two parts: a memory request (read or write) in the forward trip and a memory response (read data or write acknowledgement) in the return trip. Thus, memory access performance includes both parts. Chen et al. [139] proposed a fair-sharing method for memory access by considering the round-trip routing latency of memory accesses

to determine the priority of access contention in routers, thereby effectively achieving access fairness. However, this approach is not directly applicable to neural network memory access, where each memory access serves a unique purpose and depends on its specific operation. Additionally, the data width of each access can vary, as some memory accesses relate to input, weight, or bias data, while others are for writing and storing data.

Conventional VTA design lacks statistical evaluation component, making it necessary to perform an effective co-design to interface with the VTA architecture without altering the memory throughput when the VTA accesses the DRAM [138]. This section introduces a custom Intellectual Property (IP) co-designed with the VTA to trace and capture memory access parameters with corresponding memory access timestamps during inference runtime. The measurement is achieved using the Memory Access Tracer and Logger Unit (MATL), incorporated into the VTA, which acquires and stores each VTA module's memory execution parameters, such as the memory access address and timestamps, at runtime. These traced data are intended to populate the time-triggered system schedule after comprehensive analysis.

The Memory Access Tracer and Logger Unit (MATL), illustrated in Figure 3.3, uses a hardware-based design involving two stages: the priming stage during runtime for collecting and logging memory access information and the examination offline stage for analysing the acquired memory access traces to identify patterns and compute the schedule—the priming stage aids in acquiring measurements required to build the schedule for time-triggered memory access deployment. The primary parameters of interest to be represented in the schedule include Advanced eXtensible Interface (AXI) memory request addresses and the Memory Access Time (MAT). The MAT is a critical parameter for the schedule dispatcher of the time-triggered controller, as prior knowledge of timing is necessary to formulate a suitable scheduling algorithm [140].

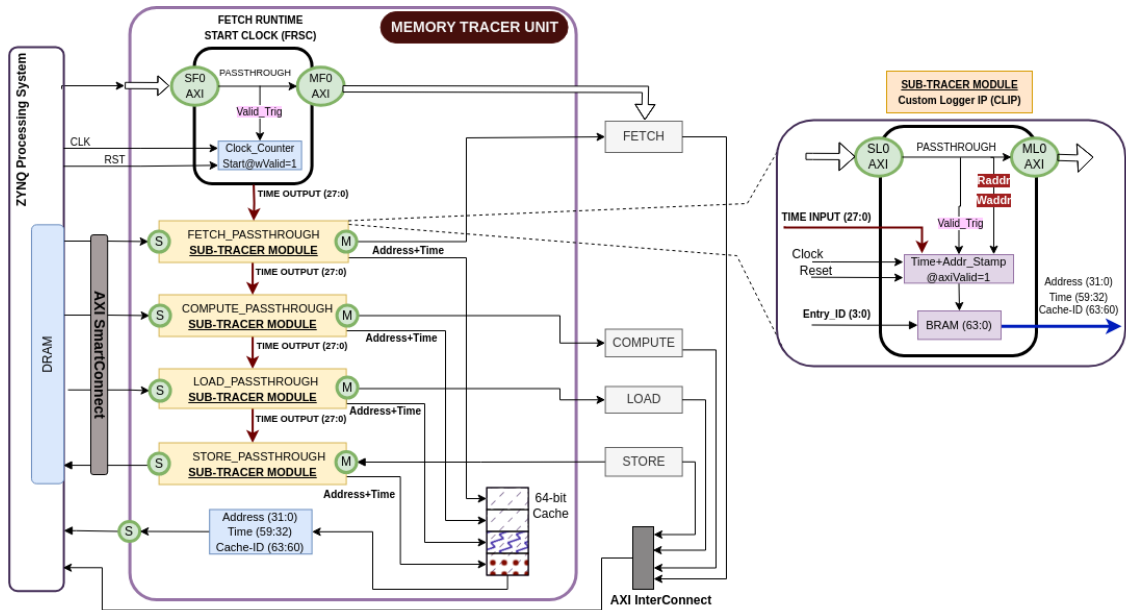


Figure 3.3: Memory Access Tracer and Logger Unit

3.3.1 Priming Phase: On-chip Memory Access Tracing of Neural Networks

Tracing memory access patterns is a challenging and complex activity [18]. As noted by [121], accurately identifying and analysing the data streams associated with memory accesses and determining the most effective address mapping scheme represents one of the most difficult aspects of this process.

This dissertation addresses these complexities by offering an effective approach to memory tracing with minimal latency without altering the original memory access throughput. This is achieved by avoiding direct or indirect interference with the AXI communication between the VTA modules and the main memory. The priming phase, represented by the Memory Access Tracer and Logger (MATL) Unit shown in Figure 3.3, is designed to trace memory accesses of the four fundamental VTA modules: Fetch, Load, Compute, and Store, each serving a distinct purpose as explained in the previous chapter. A simplified architecture of the conventional VTA and its interconnectivity with the DRAM is illustrated in Figure 3.4.

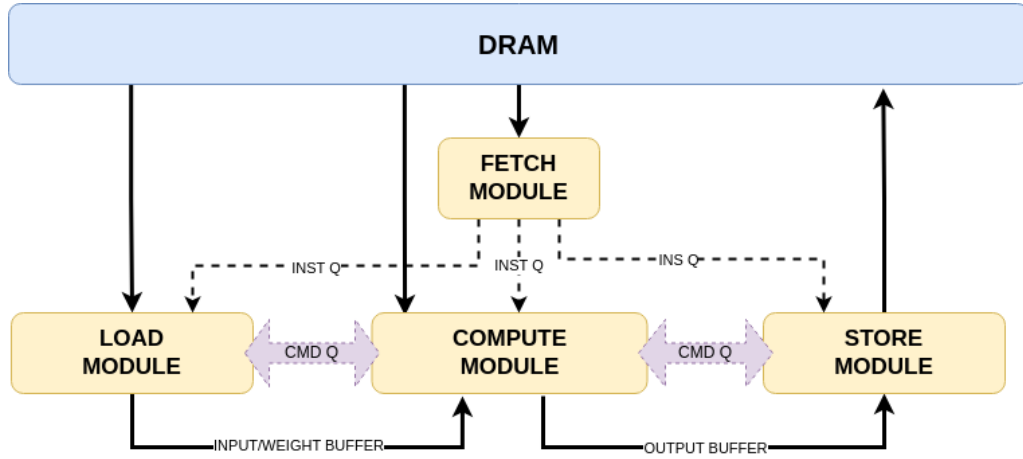


Figure 3.4: Simplified Conventional VTA Design

The MATL unit design of Figure 3.3 is an extension of the architecture in Figure 3.4, and it is made compatible with the conventional VTA architecture. It adapts the same path to the four VTA modules but is equipped with four sub-tracer modules interfaced between each VTA module and the DRAM via an AXI SmartConnect. The MATL Unit incorporates various AXI full pass-through modules implemented with a cache logging module. The system tracks the required traced parameters for each valid transaction and stores the data in Block RAM (BRAM). The design also includes an AXI cached data read module, memory-mapped to the PS high-performance HPM0 slave registers, facilitating access to the logged parameters from the PS interface. The broader co-design includes the following components:

- The Fetch Runtime Synchronized Clock (FRSC)
- Sub-tracer modules for the four VTA modules
- A BRAM cache with an Advanced eXtensible Interface (AXI) interface to the Zynq processing system.

3.3.1.1 Fetch Runtime Synchronised Clock (FRSC)

A globally synchronised clock is essential for time-based systems, enabling precise time-triggered communication and computation [135]. The global clock, also known as Global Time Base (GTB) [141], provides a consistent and unified global time for all components within the system, ensuring synchronised operations across the entire platform. When VTA modules access the main memory, an initial clock counter synchronisation is necessary to align the time logged by the MATL unit, compensating for any phase shifts [142] between the VTA’s startup and the module’s clock at the point of the first memory access. However, using a typical global clock that often starts its counter from the initial reset of the application is ineffective for an application utilising VTA due to the inability to account for time variability [143] in VTA startup tasks and the unpredictable latency of VTA operations. This discrepancy arises from several factors:

- **Software Bootstrapping and Model Fragmentation:** The inconsistency in VTA trace timing behaviour using the regular global clock concept stems from bottlenecks caused by the host system software initialisation steps—such as library loading, runtime setup, and model fragmentation—which make it unsuitable for the precise synchronisation required during VTA inference runtime. Furthermore, the VTA relies on the initialisation of Pynq drivers, background kernel threads, and software interfaces like the Remote Procedure Call (RPC) to establish communication with hardware components. These processes may introduce delays or require additional time before the hardware is fully operational, contributing to timing variability. The VTA’s runtime sequence also introduces data transfer overheads, further adding to this variability, as the Fetch module initiates transactions before other modules.
- **Initialisation Latency:** The VTA experiences initialisation overhead caused by hardware routines in setting up various components, such as configuring hardware, initialising SRAMs and caches, and setting up shared interfaces [144]. The sequential nature of these routines can lead to timing variability, especially if delays occur during the initialisation of components, such as the synchronisation of memory or peripheral interfaces.
- **System Load and Resource Contention:** Variations in the time elapsed between initialisation and the initial memory request are influenced by system load and resource contention. When multiple processes compete for the same resources, delays in processing the memory access request often occur.
- **FPGA Hardware Design and Configuration:** Latencies inherent in the physical design of the hardware, such as memory controller speed or interconnect latency, can introduce timing variations [145]. Additionally, delays in parameter configuration at startup can contribute to variability in the timing of the first memory access.

To address the issues associated with the regular global clock, which is often affected by variability during Programmable Logic (PL) initialisation (e.g., library loading, runtime setup, and model fragmentation), a custom clock synchronisation mechanism referred to as the Fetch Runtime Synchronised Clock (FRSC) is introduced to ensure precise timing in the initialisation process. An asynchronous,

event-triggered AXI message exchange from the Fetch module drives this initial clock synchronisation. As discussed in Section 2.4, all VTA transactions are initiated by the Fetch module, which begins its operations before any other module. This approach establishes a reliable global time reference for modules accessing memory during VTA operations. Unlike the regular global clock, the FRSC is specifically designed to synchronise with the start of the first task of the VTA’s Fetch module, beginning its clock counts based on the first instruction sent to the Fetch module. This method ensures precise alignment of global start times and fair reproducibility of memory access timestamps, which is crucial for maintaining consistent timing throughout VTA memory tracing at inference runtime.

The FRSC is implemented as a 28-bit clock component, providing more reliable timing behaviour compared to the regular global clock. It addresses the timing inconsistencies introduced by system initialisation and offers a more accurate representation of memory access timestamps during runtime. This synchronisation is especially beneficial in environments like the VTA, with complex initialisation sequences and varying memory access demands, ensuring consistent timing and reliable system performance.

By synchronising the clock with the VTA’s first Fetch module instruction, the FRSC ensures more precise timing for other VTA module operations, critical for real-time execution and system reliability, particularly in safety-critical applications. The FRSC improves upon the regular global clock by offering better consistency, making it suitable for time-based applications such as the TTmaVTA.

Figure 3.3 shows the FRSC module interfaced with the Sub-Tracer Modules embedded in the MATL Unit. The FRSC includes a passthrough that allows unobstructed and seamless communication between the Fetch module and the Processing System (PS). The FRSC module monitors the AXI communication bus for a valid signal and activates the clock counter only upon detecting the first valid AXI write signal. This signal serves as a synchronised clock trigger, ensuring that the clock starts uniformly across repeated runs. By using the valid AXI write signal to initiate the clock counter, the FRSC module stabilises all components, ensuring a consistent start sequence, which is critical for accurate timing and synchronisation. The clock counter within the FRSC module measures the counts. It continuously provides this time as input to the sub-tracer module for capturing and recording timestamps of memory access events during tracing.

With this operation, the FRSC module plays a crucial role in ensuring accurate timing and synchronisation across the TTmaVTA system by providing a stable and consistent global clock reference. Throughout this dissertation, the FRSC module is referred to as the global clock, as it serves as the primary timing reference for all operations within the TTmaVTA, as depicted in Figure 3.2.

3.3.1.2 Sub-Tracer Modules

The sub-tracer modules are designed to trace the memory accesses of the VTA modules, capturing their operations along with their corresponding timestamps and logging this data into a cache. There are four sub-tracer modules, each interfaced with one of the VTA modules and equipped with a specified data width that conforms to the requirements of the respective VTA module. Each sub-tracer module is a fully custom AXI IP, integrated with a Xilinx passthrough IP, incorporating AXI valid transaction trace functions. These functions capture necessary param-

ters—such as timestamps and addressstamps—during every valid signal of an AXI transaction. These parameters are captured at every AXI-valid signal, specifically at AWVALID/ARVALID or WVALID/RVALID. The Custom Logger Intellectual Property (CLIP) is an integral component of each sub-tracer module. It is an extraction from the architectural structure of the Memory Access Tracer and Logger (MATL) Unit, developed using Vivado tools with HDL and customised Xilinx passthrough IPs. The CLIP utilises Xilinx AXI Verification IPs (VIP) with the interface mode set to passthrough, the protocol configured to AXI4LITE, and the address width set to 32 bits. Other configurations, such as data width, are aligned with the VTA modules' AXI interfaces, with data width set to 64 bits or 128 bits, and transaction mode configured for read or write. The CLIP is equipped for tracing memory parameters and capturing relevant trace data. As depicted in Figure 3.5, the CLIP leverages this passthrough capability to timestamp memory access addresses while measuring the Memory Access Time (MAT) for each module's access to the memory, all without disrupting the VTA's execution process. The passthrough ensures low-latency, uninterrupted AXI signal transactions from each VTA module to the memory through the PS AXI slave HPM0 ports.

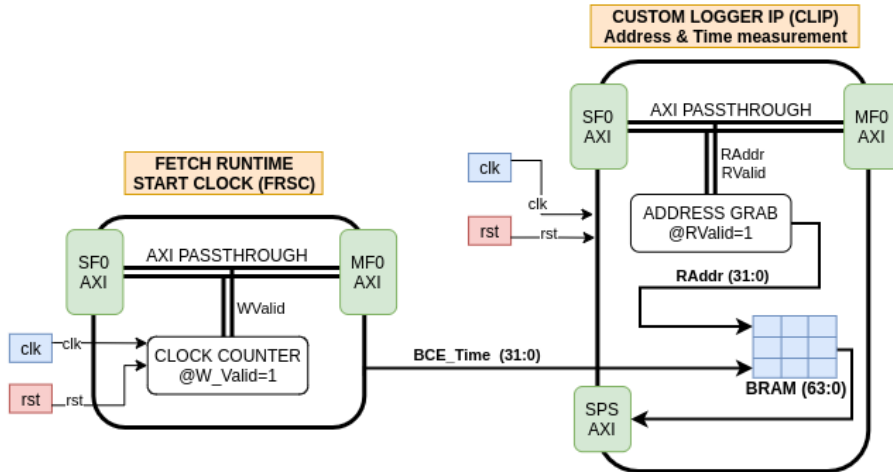


Figure 3.5: Custom Logger Intellectual Property (CLIP)

The CLIP is integrated into the VTA architecture with an additional AXI slave interface connected to the memory mapped to the PS high-performance HPM0 port. To accommodate this additional connection, the AXI interconnect is expanded with an extra AXI interface that connects the CLIP's slave port. This interface enables access from the Linux environment to the logged parameters stored in the CLIP's slave registers. The traced and logged parameters accessible from the PL interface to the DRAM include the write/read addresses and the MAT, as referenced by the FRSC.

The CLIP can trace and log four key memory access parameters: valid request, memory access address, valid response, and round-trip execution sequence. However, the first two parameters—memory access valid request and memory access address—are essential, as the memory access data constantly changes [146]. By focusing on these two parameters, the system can effectively trace memory access patterns without the variability introduced by the changing data itself.

Once valid signals are detected, the request type, addresses, and each corre-

sponding memory access timestamp (MAT) are captured and logged into the BRAM cache. The term "Memory Access Timestamp (MAT)" refers to the clock counts recorded by the Fetch Runtime Synchronised Clock (FRSC) at every valid signal. The sub-tracer module captures the write or read address (32-bit) along with the MAT (28-bit) and appends a 4-bit auto-assigned cache identifier, forming a 64-bit tracer frame that serves as input data for cache storage. These parameters are stamped at every valid AXI transaction at ARVALID and RVALID, with the traced addresses and corresponding MAT logged into the cache (BRAM).

3.3.1.3 BRAM Cache

The traced 64-bit data (comprising address, time, and cache identifier) is stored in a custom BRAM cache, beginning from the least significant bit of the cache base address to ensure proper alignment. Each traced data entry is stored as a key-value pair in the cache, corresponding to each operation. The cache depth is determined based on the BRAM sizing of the FPGA's on-chip memory using Equation 3.1. For example, the Trenz Electronics Ultrascale+ TE0802 FPGA, with 72 Kbits of BRAM, can store up to 1024 entries.

$$\text{Cache Depth} = \frac{\text{Total BRAM Size (bits)}}{\text{Width of Traced Data (bits)}} \quad (3.1)$$

The BRAM cache embedded within the MATL Unit is equipped with an AXI-slave interface which is memory-mapped to the AXI Master HPC0 port of the PS via a custom AXI Lite interface, with a designated address. This setup facilitates runtime access to the cache contents (traced data) by software applications, such as Memory-Mapped I/O (MMIO), operating on the FPGA's Linux system. The small form factor of the custom BRAM blocks allows for multiple instances, enabling parallel processing and efficient handling of traces from all four VTA modules.

A new bitstream is generated for FPGA inference after incorporating the CLIP with the VTA co-design. This bitstream can be deployed on any application running with VTA to trace memory accesses without requiring reconfiguration or alterations. The memory dynamics, including addresses and access timestamps, are captured and logged at runtime during inference. The results are retrieved via MMIO on the FPGA Linux Operating System upon completion. This marks the conclusion of the priming phase, which handles the measurement process to determine memory access patterns and behaviour. The acquired traced data is then transitioned into the offline phase, where it is retrieved from the cache for further analysis, establishing memory access patterns that will aid in developing an effective scheduling strategy.

3.3.2 Offline Phase: Examination and Analysis of Traced Memory Access Data

Following the execution of the neural network, it is crucial to promptly retrieve the logged parameters to prevent data loss due to a cache reset. One of the most efficient methods for reading and writing data between the PS and PL on Linux is through Memory-Mapped I/O (MMIO) [147], which allows the measured data on the PL to be accessible on Linux. Upon completing inference, the traced data—logged during the priming phase and stored in the BRAM cache as 64-bit entries (including

memory addresses, memory access timestamps (MAT), and cache identifiers)—are retrieved from the memory-mapped location using MMIO read commands through software applications running on Linux.

Leveraging Python productivity for Zynq (PYNQ), an efficient method is provided for interacting with the hardware. A memory buffer is allocated within the Linux virtual memory of PYNQ-Linux, enabling the CLIP-slaved AXI port, mapped to the PS, to access the physical memory via an overlay function. The MMIO’s allocation function then facilitates the creation of a contiguous memory buffer, optimising data transfers between the PL and PS.

The analysis phase involves plotting the traced data using network flow graphs and examining them to identify patterns, gaps, or system inefficiencies. These insights are crucial for generating efficient schedules and optimising memory access strategies to enhance overall system performance and predictability. A comprehensive understanding of VTA’s functionalities, particularly its modules, is imperative to interpret the traced data effectively, aiding in creating suitable schedules and efficient memory access optimisation strategies. While the MATL cache handles traces from all VTA modules, some modules contribute more significantly to memory access and latency considerations.

From a preliminary analysis of the traced data, the Fetch module accesses the DRAM only at the initialisation of the Zynq when inference is launched to load all instructions. After this initial operation, the Fetch module then controls the other three modules by allowing their operations to proceed based on the commands in their respective dependent queues. Consequently, in the subsequent design phase, the Fetch module is excluded from further analysis, as it accesses the main memory only once at the start of the inference process. Additionally, while the Compute module, including the accumulators and micro-operations, accesses the DRAM at repetitive intervals, its memory access durations are minimal, as estimated using Equation 3.2. This equation computes the contribution of each module to memory access, quantifying how much each module contributes to overall memory access and latency.

$$C_i = \frac{\sum_{j=1}^{N_i} T_{ij} \times O_{ij}}{\sum_{k=1}^M \sum_{l=1}^{N_k} T_{kl} \times O_{kl}} \times 100 \quad (3.2)$$

where:

- C_i is the contribution percentage of module i to the total memory access.
- T_{ij} represents the total access time for operation j in module i .
- O_{ij} is the frequency of operation j in module i .
- N_i is the number of distinct operations in module i .
- M is the total number of modules.
- N_k is the number of distinct operations in module k (where k ranges over all modules).

This formula calculates each module’s weighted contribution based on its access times and operation frequencies relative to the total access times and operation frequencies across all modules. It provides a percentage indicating how much each module contributes to overall memory access.

The computation based on Equation 3.2 shows that the total memory access duration accounted for by the Compute module is approximately 5% of the duration

required for a load input. As a result, the extended analysis excludes the Fetch module due to its minimal contribution after initialisation and does not delve deeply into the Compute module, as its contribution to total DRAM memory access is nearly negligible. The focus shifts to the Load and Store modules, which account for most memory transactions. These modules represent about 95% of memory accesses and significantly influence memory latency. The trace analysis of these modules (Load and Store) is concentrated on understanding their memory access patterns, given their impact on system performance.

3.4 Memory Access Plot Representation and Analysis

The convolution operations in CNNs intensify challenges in accelerator memory access, such as row conflicts that increase access latency [121], impacting the overall efficiency of neural network accelerators. These challenges emphasise the need to trace memory access patterns to identify opportunities to optimise performance. By analysing these patterns, a deeper understanding of how memory access affects overall performance is achieved, allowing for more efficient memory management strategies and optimised scheduling of memory accesses in neural network accelerators.

This section delves deeper into the trace data plot using a network flow graph, which includes nodes, directional arrows, and branches. It also extends the trace analysis by providing more in-depth insights into memory access behaviour and offering improved strategies for optimisation and scheduling techniques aimed at streamlining memory accesses and eliminating inefficiencies, such as contention and dependency-related bottlenecks identified by researchers in previous studies.

As previously outlined in Section 3.3, and depicted in the system model in Figure 3.2, the proposed time-triggered memory access architectural design operates optimally with a predefined schedule, a crucial element of time-triggered systems that requires prior knowledge of memory access patterns, including addressing and the sequence of operations [138]. Memory access tracing in neural networks provides valuable insights into the interaction between computational units and memory subsystems.

This section begins with a thorough examination and analysis of the memory access traces gathered from the execution of neural networks, such as ResNet-18, on the VTA to identify memory access patterns. Various trace data, including parameters such as memory access timestamp, address, operation type (read or write), and sequence of operations collected in Section 3.3, are analysed using operational plots, such as network flow graphs and other analytical software simulations. This tracing data is fundamental for constructing a detailed **Network Flow Graph** representation of memory operations to visually depict the memory access patterns. These insights aid in optimising memory accesses and conceptualising adaptable scheduling techniques to mitigate latency, reduce idle memory phases, and handle dependency-related stalls.

3.4.1 Network Flow Graph Representation of Memory Accesses

During the memory access tracing phase in Section 3.3, critical parameters are captured; this involves tracking parameters such as access times, memory addresses, and operation types (read or write). The traced memory access data can be modelled as a **Network Flow Graph**. In this graph, each node represents individual memory access events (read/write), and each directed edge (branch) represents the dependency between consecutive accesses. Arrows depict the operation's direction and represent the data flow between nodes. The key graph attributes include:

Node (N): A node represents a single memory access event, which can be characterised by the following parameters:

- t : The memory access timestamp.
- a : The memory access request address.
- op : The type of operation (read or write).
- seq : The sequence number of the operation, determining its relative order.

Each node N_i is characterised by the tuple:

$$N_i = (t_i, a_i, op_i, seq_i) \quad (3.3)$$

Where:

- i represents the access index.
- N_i represents a node corresponding to the i -th memory access event.
- t_i is the timestamp of the i -th access.
- a_i is the address accessed.
- op_i indicates the operation type (read or write).
- seq_i is the sequence number of the operation.

Edge (E): An edge represents the relationship between two consecutive memory accesses:

- *Dependencies*: These dependencies are created when one memory access is dependent on the completion of another. For example, a write operation must be completed before a subsequent read operation can access the same memory location.
- *Arrows*: The arrows on the graph indicate the flow of memory operations between nodes.

Each edge, E_{ij} , represents a dependency and is associated with a weight w_{ij} , which represents the latency or delay introduced by the dependency.

Where: E_{ij} represents a directed edge from node i to node j , indicating that access i must complete before access j can start.

The mathematical representation of the memory access graph is given by:

$$\text{Memory Access Graph} = G(N, E) \quad (3.4)$$

Where N is the set of nodes and E is the set of directed edges representing dependencies between memory operations.

The expanded form of the memory access graph derived from Equation 3.4 and Equation 3.3 can be represented as:

$$G(N, E) = \left(\bigcup_i N_i(t_i, a_i, op_i, seq_i), \bigcup_j E_j(N_{src(j)}, N_{dest(j)}) \right) \quad (3.5)$$

Where:

- $N_i(t_i, a_i, op_i, seq_i)$ represents the i -th node.
- $E_j(N_{src(j)}, N_{dest(j)})$ represents the j -th directed edge with source node $N_{src(j)}$ and destination node $N_{dest(j)}$.

In practice, the graph structure may involve multiple levels of dependencies, such as:

- **RAW Dependencies:** Read-after-write dependencies, where a read operation must wait for the preceding write to complete, predominant in the VTA traced data.
- **WAW Dependencies:** Write-after-write dependencies, where a write operation must wait for a previous write to complete to the same address.
- **WAR Dependencies:** Write-after-read dependencies, where a write must wait for a previous read to the same address.

Each of these dependencies adds complexity to the scheduling problem. A typical Network Flow Graph can be represented as follows:

$$G(N, E) = (\{N_1, N_2, \dots, N_k\}, \{E_{12}, E_{23}, \dots, E_{k-1,k}\})$$

Where k is the total number of memory access events.

Hence a more detailed representation of the Network Flow Graph is given by:

$$G(N, E) = \left(\bigcup_{i=1}^k N_i(t_i, a_i, op_i, seq_i), \bigcup_{j=1}^m E_j(N_{src(j)}, N_{dest(j)}, w_{ij}) \right) \quad (3.6)$$

Where:

- $N_i(t_i, a_i, op_i, seq_i)$ represents the i -th node, characterised by:
 - t_i : Timestamp of the i -th access.
 - a_i : Address accessed.
 - op_i : Operation type (read or write).
 - seq_i : Sequence number of the operation.
- $E_j(N_{src(j)}, N_{dest(j)}, w_{ij})$ represents the j -th directed edge, characterised by:
 - $N_{src(j)}$: Source node of the edge.
 - $N_{dest(j)}$: Destination node of the edge.
 - w_{ij} : Weight associated with the edge, representing the latency or delay.

In this context:

- k is the total number of nodes (memory access events).
- m is the total number of directed edges (dependencies between accesses).

The computations are carried out using the formula in Equation 3.6 to analyse the traced memory access data. The resulting memory access graph, depicted in Figure 3.6, is generated using a Python-based software tool within a Code Execution Workspace, which applies the principles outlined in Equation 3.6.

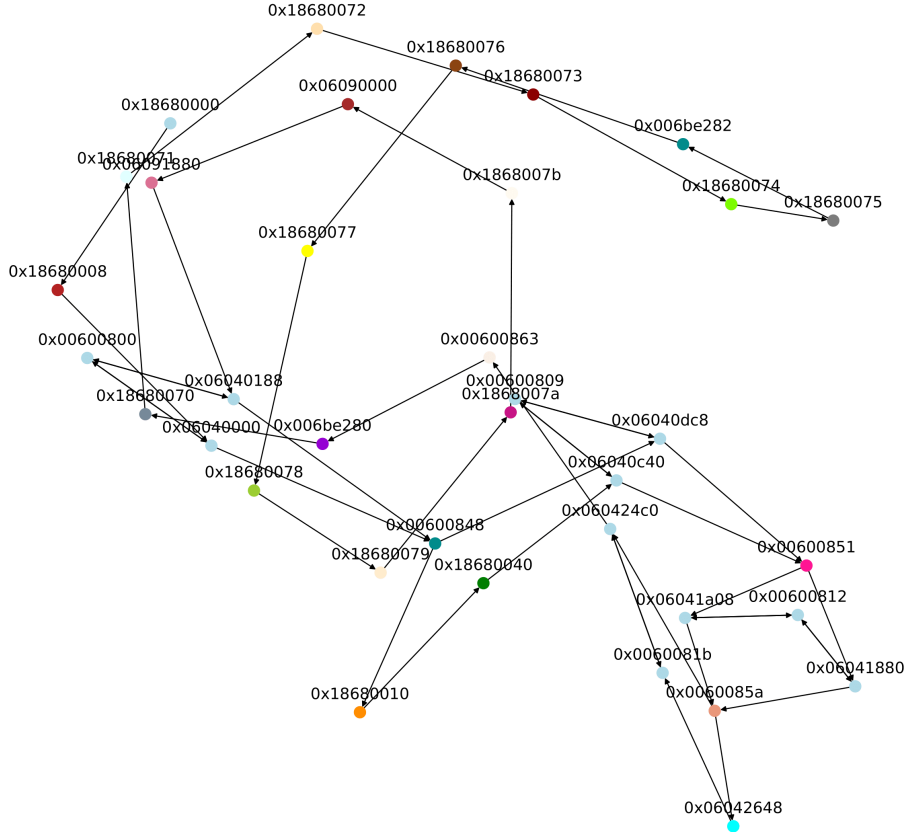


Figure 3.6: VTA Layer-wise Flow Graph Representation

The network flow graph illustrates memory access patterns across multiple layers, showcasing the interaction between **STORE** and **LOAD** operations. Each **STORE** node represents a memory write operation, while **LOAD** nodes (**LD**) denote read operations from the same memory address. The labels **LDa**, **LDb**, **LDC**, etc., indicate successive reads from a specific **STORE**, representing multiple reads from the same write.

Directed edges between nodes highlight data dependencies, with **RD** marking the read path from **STORE** to its corresponding **LOAD** operations. The thicker red dashed edges represent overwrite behaviour, where multiple writes occur to the same memory location, effectively replacing previous stored data.

The graph is organised in layers, each corresponding to a different memory access cycle or phase. These layers reflect the temporal flow of data between write and read operations, providing a structured view of memory utilisation and read-write

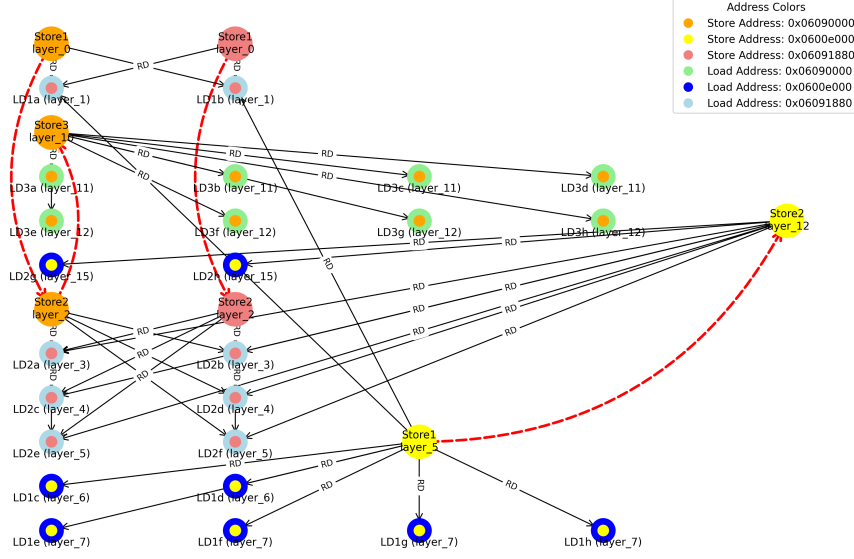


Figure 3.7: VTA memory pattern Address Graph Representation

contention. The use of distinct colours for nodes highlights memory addresses, emphasising the differentiation between STORE and LOAD operations across the various memory layers.

3.4.2 Memory Access Pattern and Latency Analysis

Memory performance is closely linked to the access patterns generated by an application [148]. In computationally intensive tasks, such as neural network operations on accelerators, the memory system often experiences a high demand for bandwidth. For instance, in a CNN-based accelerator like the VTA, typical memory access patterns during CNN computations account for more than 95% of all memory operations in image processing tasks [149].

In such computations, the accelerator starts by obtaining feature maps and weights from the host machine and storing them in the main memory during initialisation. The FPGA’s computing unit then retrieves the feature maps and weights from memory to perform calculations. This process is iterative: the computing unit fetches feature maps, executes calculations, and stores intermediate results back in the main memory. After several computational iterations, the final inference results are transmitted back to the host machine.

These computations necessitate a thorough analysis to establish memory access patterns. This section provides a comprehensive analysis of memory access patterns and their associated latencies, evaluated from the traced data to understand the intricate dynamics of memory operations within the VTA. By examining these patterns, the underlying factors influencing memory performance, particularly concerning dependencies, cache behaviour, and the sequence of operations are identified.

To effectively carry out the trace analysis, a set of key parameters are selected that provide crucial insights into the behaviour and performance of memory operations. These parameters serve as the foundation for the traced analysis, allowing for quantification and evaluation of various aspects of memory access. The key parameters chosen for analysis are:

- **Memory Access Time:** This parameter records the precise timestamp at which a memory access occurs during tracing. It provides a fundamental metric for tracking the timing of each operation, critical for assessing overall memory access latency.
- **Address:** The memory address being accessed is essential for identifying potential data hazards, such as read-after-write (RAW) or write-after-write (WAW) conflicts. Additionally, memory addresses are instrumental in determining memory access, influencing factors such as cache hits and misses, and directly impacting performance.
- **Type of Operation (Read/Write):** Memory operations are categorised into reads and writes, each exhibiting distinct timing behaviours. Reads and writes often differ significantly in terms of latency, with writes typically incurring additional delays due to coherence and synchronisation mechanisms within the memory hierarchy.
- **Sequence of Operations:** Ordering memory operations is crucial for maintaining consistency, particularly when handling dependencies. A detailed study of the VTA demonstrates a sequential ordering of memory operations. The sequence number associated with each operation is used to elucidate the memory access pattern, representing the flow of memory accesses, which is vital for analysing dependencies and operation order.
- **Memory Access Time Delay:** This parameter captures the delay between the issuance of a memory access request and its completion. The access time delay is influenced by factors such as cache hits or misses, contention, and whether the operation is a read or a write. Additionally, delays in memory access can be caused by row conflicts during convolution computations. It is quantified as the ratio of the current system running time to the system's full row buffer hit running time and can be calculated using the following formula:

$$\frac{T_{CL} \times \text{No. Hits} + (T_{CL} + T_{RCD} + T_{RP}) \times \text{No. Misses}}{T_{CL} \times (\text{No. Address Sequences} - 1)} \quad (3.7)$$

(Source: [121])

Where:

- T_{CL} : Column Access Strobe (CAS) Latency – the time interval between issuing a read command and the availability of data from memory.
- T_{RCD} : Row to Column Delay – the delay between row activation and the subsequent column access.
- T_{RP} : Row Precharge Time – the time required to close an active row before accessing a different row.
- No. Hits: The number of memory accesses that successfully retrieve data from the cache.
- No. Misses: The number of memory accesses that require fetching data from the main memory due to cache misses.
- No. Address Sequences: The total number of unique memory access requests.

The T_{CL} , also represented as T_{latency} , does not account for factors such as memory contention, which influence access time and lead to additional delays. The overall access delay can be expressed as:

$$T_{\text{access delay}} = T_{\text{queue}} + T_{\text{latency}} \quad (3.8)$$

Here, T_{queue} represents the time spent waiting due to contention, while T_{latency} denotes the inherent latency of the memory operation, expressed as T_{CL} . The resulting equation is:

$$T_{\text{access delay}} = T_{\text{queue}} + \frac{T_{CL} \times \text{No. Hits} + (T_{CL} + T_{RCD} + T_{RP}) \times \text{No. Misses}}{T_{CL} \times (\text{No. Address Sequences} - 1)} \quad (3.9)$$

- **Memory Access Duration:** The duration measures the time elapsed from the start of a memory access (issuance) to its final completion. This duration varies based on multiple factors, including the type of operation, system load, and memory hierarchy. It is a critical metric for understanding how efficiently the memory system is performing under different operational conditions and can be defined as:

$$T_{\text{duration}} = T_{\text{completion}} - T_{\text{issuance}} \quad (3.10)$$

Where:

- T_{duration} is the memory access duration.
- $T_{\text{completion}}$ is the timestamp when the memory access operation is completed.
- T_{issuance} is the timestamp when the memory access operation is initiated.

This formula directly measures the latency involved in each memory access, helping to identify bottlenecks and optimise memory performance.

Following the selection of these parameters, the analysis proceeds by investigating their interrelationships and impacts on overall system performance. This includes examining the effect of memory operation dependencies on subsequent memory operations, analysing the relationships between memory access pairs, and evaluating the sequence of operations in relation to their predecessors, such as the reference of a Store operation to a previous Store operation. Additionally, the time duration and variation between similar memory operations, such as load inputs, are analysed to identify patterns or irregularities. The influence of non-memory operational phases on memory access performance is also considered, particularly in cases where non-memory operations, such as computational tasks, introduce dependencies that affect memory execution timing.

The analysis employs a custom Python-based software tool within a Code Execution Workspace designed to analyse the neural network trace data and model memory operations within the system. By systematically utilising the software tool to study the relationships and impacts of these factors, deeper insights into the

behaviour and performance of memory access patterns are gathered. The analysis focuses on the following key areas:

1. **Effect of Memory Operation Dependencies on Subsequent Operations:** Memory operations often exhibit dependencies where the completion of one operation directly influences the initiation of the next. For instance, a read operation might be delayed until a prior write operation is completed to ensure data consistency. Analysing these dependencies can help identify bottlenecks and optimise the sequence of operations to reduce latency or memory idleness.
2. **Sequence and Relationship Between Memory Access Pairs and Predecessors' Deployment:** The sequence in which memory operations are deployed plays a critical role in system performance. The analysis examines how memory access pairs, such as consecutive read/write operations, are ordered and how their deployment sequence affects latency. Understanding the impact of predecessor operations on subsequent accesses can reveal opportunities for reordering or parallelising tasks to improve efficiency.
3. **Time Duration and Access Variation of Similar Memory Operations:** Different types of memory operations, such as Load input, Load weight, and Store, may exhibit varying time durations and access latencies depending on system conditions. The analysis identifies patterns that can help predict and mitigate performance issues by comparing these variations. For instance, frequent cache misses during load operations, evident in longer execution times, suggest the need for cache optimisation strategies.
4. **Non-Memory Operational Phases and their Impact on Memory Access Performance:** Memory operations do not occur in isolation; they often interact with non-memory instructions, such as computational tasks. These interactions can introduce additional delays or dependencies that impact memory access performance. The analysis investigates how non-memory operational phases influence memory operations, particularly in scenarios where computational tasks must be completed before certain memory accesses can proceed. Understanding these interactions is crucial for optimising the overall workflow.
5. **Identification of Memory Operations Contributing to the Highest Latencies:** Some memory operations may consistently contribute to higher latencies due to factors such as contention, synchronisation, or complex dependencies. By identifying these operations, alongside reevaluating traced data using Equation 3.2, the analysis can target specific areas for optimisation, such as improving memory allocation strategies, enhancing caching mechanisms, or redesigning the sequence of operations to minimise delays.

These selected aspects of the analysis offer a detailed examination of prevalent issues in memory access patterns and their impact on performance. The identified memory access patterns illustrate a sequence of operations transitioning from LOAD micro-operations to GEMM computations, followed by LOAD input, LOAD weight,

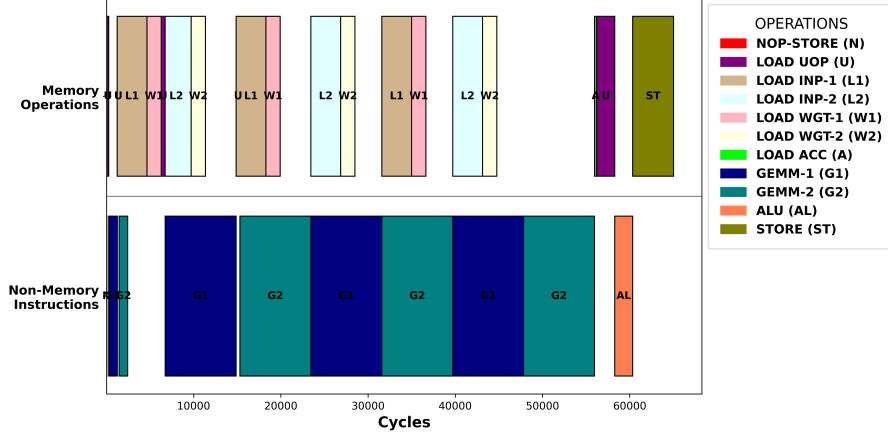


Figure 3.8: Typical VTA Operation pattern

additional LOAD micro-operations, LOAD accumulators, ALU computations, and concluding with store operations and a FINISH signal, as depicted in Figure 3.8.

These memory operations often interface with non-memory operations, such as the computation phase depicted in Figure 3.8, creating dependencies that influence the timing and order of memory accesses. For example, a 'GEMM' computational task may need to be completed before a Load input memory operation can proceed, introducing a dependency. Similarly, a Load weight operation might depend on the completion of a Store operation in certain layers of the network, necessitating that the Load weight waits for the Store to finish. Understanding these intricate relationships, the impacts of each trace parameter on memory access patterns, and the associated latencies of each memory access is crucial for devising an appropriate scheduling technique and developing strategies to optimise memory operations with reduced latency. This is particularly important for enhancing system efficiency in large neural networks with complex dependencies and varying memory access behaviours.

3.4.3 Insight into Memory Access Behaviour

After conducting a thorough trace analysis utilising various tools, along with the plots and data representations discussed in Sections 3.4.1 and 3.4.2, several key observations are made, revealing distinct behaviours in various memory operations. On a broader scale, three significant patterns are identified:

- (A) **Overlapping Operations:** Overlapping memory operations occur when multiple memory accesses attempt to use the memory bus concurrently, competing for the same limited resources, leading to contention and inefficiencies. These overlaps create memory bottlenecks that affect overall memory throughput. Overlaps are detected by analysing the start and end times of memory accesses. When two or more accesses to different addresses overlap in execution time, it indicates contention at the memory bus.
- (B) **Idle Memory Phases:** Idle memory phases arise when operations that initially access the bus are stalled, waiting for dependent operations to complete.

For example, a read operation may be delayed until a preceding write operation to the same address finishes. Idle phases can be identified by detecting gaps, free bus access periods, in the memory operation timeline during the neural network run—periods where no memory access occurs despite queued operations. These phases often result from dependency waiting, such as a read-after-write (RAW) hazard, where a read must wait for a preceding write to the same address to complete, before proceeding for a read.

- (C) **Frequent Repetitive Memory Accesses:** Another inefficiency observed is the redundant activity by repeated memory access to the same memory locations within short intervals. These frequent data fetches keep the memory bus active, continually retrieving the same data from time to time. For instance, a write operation in one neural network layer may lead to multiple read instances in the same or subsequent layers, which implies multiple memory accesses, contributing largely to memory access inefficiencies. This repetitive access monopolises the memory bus, reducing availability for other memory operations and affecting overall memory throughput, especially when these instances occur in quick succession within the same layer.

Memory access timing and the sequencing of operations are critical for optimising memory access efficiency and, consequently, for generating an optimised schedule. The first observation, (A), poses a significant risk to the timing of executions, impacting the scheduling process, especially in time-triggered systems whose schedules require unique time access for each instance. This concern is addressed in this chapter, focusing on eliminating overlaps between memory operations. The latter two observations, (B) and (C), which relate more to memory access optimisations aimed at improving memory throughput, will be explored in Chapter 4, along with strategies to address these issues.

In this context, "overlapping operations" refers to memory contention caused by simultaneous memory requests. This situation arises when different memory operations attempt to access the memory concurrently. The plot in Figure 3.9 illustrates a single block operation of ResNet-18 Layer 1, highlighting memory contention, evidenced by the overlap marked in red by the store modules. This notable example of memory contention shows a store operation (highlighted in red) attempting to write data to memory while other operations, such as reading input and retrieving weights, are already occupying the memory bus. These contentions result in memory interference and congestion on the memory bus [150].

The identified memory access patterns from the traced data are heavily influenced by contention from multiple concurrent memory requests, contributing to delays and unfair memory access [139]. Such overlapping operations within the VTA can reduce system efficiency. To mitigate this issue, an analytical approach is proposed that enforces sequential, time-based execution, ensuring that one memory operation completes before the next request begins. This method helps deliver inference results promptly while avoiding memory interference that could cause execution delays [150]. The procedure begins by analysing memory operations' start and finish times to identify, extract, and address overlaps.

Let the start time of operation i be denoted as $t_{i,\text{start}}$, and the finish time as $t_{i,\text{finish}}$. The finish time is calculated using the duration d_i of operation i , such that:

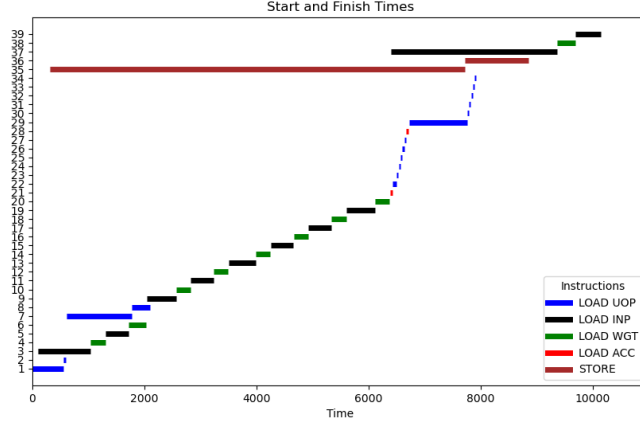


Figure 3.9: Memory Access Contention in ResNet-18 Block 1, Layer 1

$$t_{i,\text{finish}} = t_{i,\text{start}} + d_i \quad (3.11)$$

For two operations i and j to overlap, the following condition must hold:

$$t_{i,\text{start}} < t_{j,\text{finish}} \text{ and } t_{j,\text{start}} < t_{i,\text{finish}} \quad (3.12)$$

To ensure that no two memory accesses overlap, operations must be adjusted based on their execution durations and dependencies. This adjustment can be formulated as a **Constraint Satisfaction Problem (CSP)**, where the constraints are based on the start and finish times of the operations.

The variables and constraints are defined as follows:

- $t_{i,\text{start}}$: Start time of operation i .
- $t_{i,\text{finish}} = t_{i,\text{start}} + d_i$: Finish time of operation i .
- d_i : Duration of operation i .
- $t_{i,\text{start}} < t_{j,\text{finish}}$ and $t_{j,\text{start}} < t_{i,\text{finish}}$: Overlap condition between operations i and j .

To prevent overlaps, constraints must be applied to adjust the memory access timestamps so that the finish time of the current operation becomes the start time of the next operation where overlaps might occur. This adjustment respects all necessary conditions, such as dependencies and predecessor accesses. The overlap condition is managed using two approaches:

- Realignment Strategy:** This involves adjusting operation timings to ensure that $t_{i,\text{finish}} \leq t_{j,\text{start}}$ or $t_{j,\text{finish}} \leq t_{i,\text{start}}$ for all operations i and j . This strategy realigns operations to eliminate overlaps.
- CSP Formulation:** In this approach, the scheduling problem is modelled as a CSP. Variables represent the start and finish times, with constraints ensuring that no overlaps occur between operations. This method provides a formal framework to manage the constraints systematically.

Algorithm 1 Memory Access Overlap Detection

```

1: procedure DETECTOVERLAP(List of Memory Accesses)
2:   Initialise overlap_list  $\leftarrow$  []
3:   for each access  $A_i$  in memory_accesses do
4:     for each subsequent access  $A_j$  do
5:       if  $A_i$  and  $A_j$  overlap in time then
6:         Add  $(A_i, A_j)$  to overlap_list
7:       end if
8:     end for
9:   end for
10:  return overlap_list
11: end procedure

```

A combination of the **Realignment Strategy** and **CSP Formulation** proves effective in managing the observed behaviours. The pseudocode in Algorithm 1 outlines the steps to detect and manage overlaps caused by concurrent memory accesses, based on their timestamp information.

3.5 Time-Triggered Scheduling

After analysing the traced memory access patterns and identifying and addressing inefficiencies such as overlapping operations, the next step is establishing a scheduling strategy where these issues are systematically mitigated. The primary objective of this optimisation is to enhance the predictability and efficiency of memory access within the system. A well-structured time-triggered schedule provides a deterministic framework, ensuring that each memory operation is executed when required without contention while ensuring predictability. This section outlines the methodology for creating such a schedule, which is integral to achieving the desired performance and timing guarantees for neural network processing on the VTA.

To produce a suitable schedule for the TTmaVTA that can guarantee predictable memory access performance, this dissertation proposes a time-triggered scheduling mechanism. This mechanism pre-allocates time slots for each memory access operation based on the traced data, memory access patterns, and trace analysis, accounting for the optimised MAT of each operation. The time-triggered scheduling approach ensures that memory accesses are executed at precise time intervals, offering streamlined execution by incorporating Algorithm 1, thereby reducing the likelihood of memory contention.

Software-based scheduling techniques are adopted to generate the time-triggered schedule, where each memory access is assigned a specific time slot based on its execution order, considering dependencies, the overall memory access flow sequence, and the MAT. The MAT table computed in the first procedure of Algorithm 2 is determined by analysing the maximum latencies observed across multiple runs. These latencies are used to refine and optimise the MAT table, ensuring the schedule is robust against timing variations during execution. The generated schedule ensures that operations are not only executed within their allocated time slots but are also temporally aligned, enhancing the predictability of the system and preventing overlapping operations that could lead to contention. By incorporating the optimised MAT into the scheduling process, the system can guarantee that each operation completes within its allocated time slot, further enhancing predictability.

Furthermore, the scheduling process incorporates the steps outlined in Algorithm 1 to detect potential contentions and adjust memory access times, resulting in a schedule free from multiple concurrent accesses. Software-based scheduling techniques aim to organise the execution of operations according to Algorithm 2 to manage memory contention and maximise the use of available resources. Algorithm 2 outlines the steps taken to generate a suitable time-triggered schedule for a neural network running on VTA based on the traced memory access data and MAT analysis.

Algorithm 2 Time-Triggered Scheduling

```

1: procedure TIMETRIGGEREDSCHEDULING(Memory Access Trace Data)
2:   Initialise Schedule  $S \leftarrow \{\}$ 
3:   Collate Memory Access Timestamps from 1000 Loop Runs
4:   Initialise MAT_list  $\leftarrow []$ 
5:   for each memory access operation type  $Op_i$  do
6:     Identify timestamps for  $Op_i$  across all loops
7:     Compute the time differences for  $Op_i$  operations
8:     Determine the maximum time difference for  $Op_i$ 
9:     Store this value in MAT_list for  $Op_i$ 
10:  end for
11:  Sort Accesses by Sequence Number
12:  Initialise overlap_list  $\leftarrow []$ 
13:  for each access  $A_i$  in memory_accesses do
14:    for each subsequent access  $A_j$  do
15:      Compute finish time for  $A_i$ :  $t_{i,finish} = t_{i,start} + d_i$ 
16:      Compute finish time for  $A_j$ :  $t_{j,finish} = t_{j,start} + d_j$ 
17:      if  $t_{i,start} < t_{j,finish}$  and  $t_{j,start} < t_{i,finish}$  then
18:        Add  $(A_i, A_j)$  to overlap_list
19:      end if
20:    end for
21:  end for
22:  for each access  $A_i$  in sorted order do
23:    Identify operation type  $Op_i$ 
24:    Compute initial time slot  $T_i$  using MAT for  $Op_i$ 
25:    Compute  $finish\_time_i$  as  $T_i + d_i$ 
26:    if  $A_i$  is in overlap_list then
27:      Adjust  $T_i$  to start after the finish time of the overlapping  $A_j$ 
28:      Recompute  $finish\_time_i$  as  $T_i + d_i$ 
29:    end if
30:    if previous operations not completed then
31:      Delay memory access until previous operation completes
32:    if Dependency Detected then
33:      Delay access until dependent operation completes
34:    end if
35:  end if
36:  for each memory access operation, form a 64-bit Schedule Entry do
37:    2-bit entry point identifier
38:    32-bit memory address  $A_i.addr$ 
39:    28-bit time slot  $T_i$ 
40:    2-bit operation identifier  $Op_i$ 
41:  end for
42:  Store the formed 64-bit Schedule Entry in  $S$ 
43: end for
44: return Schedule  $S$ 
45: end procedure

```

The schedule consist of 64-bit entries containing:

- 2-bit entry point identifier (denoting the entry condition for the access),
- 32-bit memory address,
- 28-bit time slot,
- 2-bit operation identifier (read [load, compute] or write [store]).

Incorporating the various techniques into Algorithm 2 contributes to generating a schedule optimised for enhancing the performance of neural networks by ensuring predictability, as each operation executes with timing guarantees. By employing the scheduling algorithm in conjunction with a mapping scheduling formula and considering the memory access time, T_{access} , along with task dependencies, the schedule is computed as shown in Equation 3.13:

$$\text{Schedule Entry}_i(E_i) = (\text{Entry}_i, \text{Address}_i, T_i, \text{Op}_i) \quad (3.13)$$

Where:

- Entry_i represents the 2-bit entry point identifier,
- Address_i represents the 32-bit memory address,
- T_i represents the 28-bit time slot,
- Op_i represents the 2-bit operation type identifier (read/write).

Thus, each entry E_i can be viewed as a tuple:

$$E_i = (\text{entry point identifier, memory address, time slot, operation identifier})$$

The overall schedule can be represented as a set of 64-bit entries:

$$S = \{E_1, E_2, \dots, E_n\} \quad (3.14)$$

Where each entry E_i corresponds to a specific memory access.

Given that each schedule entry E_i is defined by Equation 3.13, the overall schedule S can be expressed as a set of such 64-bit entries using Equation 3.15 as:

$$S = \{(\text{Ent}_1, \text{Add}_1, T_1, \text{Op}_1), (\text{Ent}_2, \text{Add}_2, T_2, \text{Op}_2), \dots, (\text{Ent}_n, \text{Add}_n, T_n, \text{Op}_n)\} \quad (3.15)$$

Where each tuple $(\text{Entry}_i, \text{Address}_i, T_i, \text{Op}_i)$ represents a specific memory access operation with its respective entry point identifier, memory address, time slot, and operation identifier corresponding to a Schedule Entry E_i , capturing all necessary scheduling details.

Schedule Generation

The last phase of memory tracing is the generation of a suitable schedule for the time-triggered architecture aided by the scheduling algorithm. Rooted in the data recorded during the memory access tracing, the schedule is generated based on this traced data analysis, identified memory addressing patterns, and known memory access timing sequences utilising a Python-based software tool within a Code Execution Workspace, leveraging Algorithm 2. The scheduling pattern of the 64-bit schedule comprises four slices: the first 2-bit segment is the next entry execution identifier, which starts from $i = 0$, incrementing by $i + 1$ and then followed by a

32-bit address request slice, a 24-bit memory access timestamp slice, and the last 2-bit as cache slot identifier, which identifies the cache location to write/read the retrieved data.

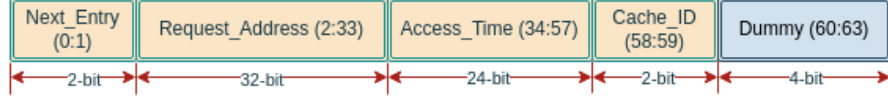


Figure 3.10: Schedule Frame

On initialising the software tool, it takes the various traced parameters, applying Algorithm 2 to align the entry sequentially based on the execution order, adjust the entry MAT and finish MAT of each operation based on the computed duration, ensuring an operation finishes before the start of another while taking reference of the predecessors and the dependencies to avoid collision. Finally, the optimised memory addresses, memory access timestamps (MATs), and task execution orders are sorted and profiled to formulate/generate a suitable schedule file, which is preloaded to the FPGA during the bitstream initialisation for use in the online phase of the TTmaVTA.

3.6 System Application Model: Time-Triggered Memory Access VTA (TTmaVTA)

Neural network accelerators, such as the Versatile Tensor Accelerator (VTA), are increasingly vital in deep learning, providing the computational capacity required for advanced AI applications. As AI demand increases, ongoing innovations in accelerator design are pivotal in advancing machine learning capabilities. The VTA, an open-source accelerator, has been widely applied and is instrumental in several deep learning acceleration tasks, offering tailored solutions for flexible prototyping on FPGAs.

Despite its advantages, the VTA architecture faces challenges related to memory management and timing requirements, which can impact its performance and predictability, rendering it unsuitable for safety-critical applications. To address these challenges, this dissertation proposes a co-design for the VTA, presenting opportunities for optimising memory throughput, enhancing predictability, and ensuring optimal resource utilisation. This section introduces a time-based memory access system integrating the VTA with a time-triggered hardware co-design framework. This modified architecture, named Time-Triggered Memory Access VTA (TTmaVTA), leverages the hardware portability of the VTA with optimisation techniques to improve memory access while ensuring predictability through a time-triggered framework.

The TTmaVTA, depicted in Figure 3.11, integrates a Time-Triggered Memory Access Controller (TTmaC) with the VTA modules via an AXI interconnect bus. This co-design leverages the Fetch Runtime Synchronised Clock (FRSC) discussed in Section 3.3.1.1, which serves as the global clock and is central to the TTmaVTA's operation. The FRSC interacts with the schedule preloaded into the TTmaVTA during inference at initialisation, facilitating time-based, preemptive memory access.

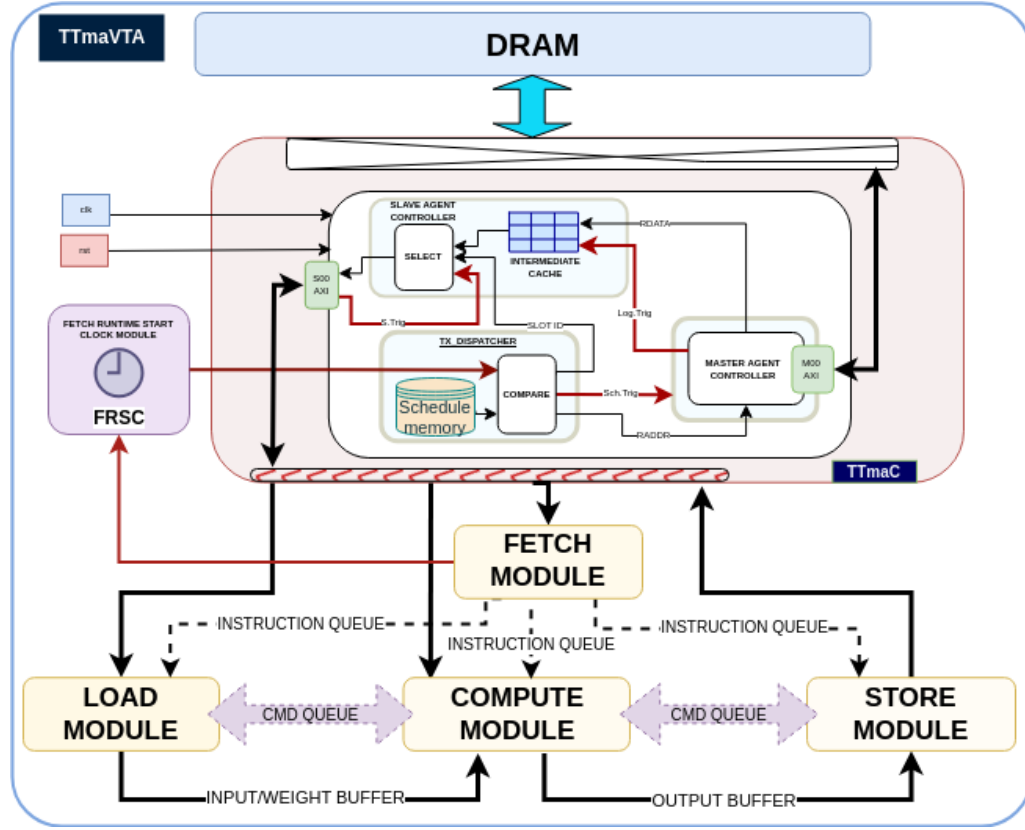


Figure 3.11: Time-Triggered Memory Access VTA (TTmaVTA) Co-Design

The TTmaVTA operates by synchronising the current global time provided by the FRSC with the time slices defined in the schedule from Section 3.5, generated from the traced data analysis. As depicted by the TTmaVTA task coding state-machine in Figure 15 (see Appendix A), once a match is identified between the global clock and a schedule entry, the TTmaVTA’s dispatcher component triggers a data retrieval from memory at the address specified by the corresponding address slice in the schedule. The retrieved data is temporarily loaded into an intermediate cache embedded in the TTmaC, allowing for momentary storage for easy access by the VTA modules on request. This time-triggered memory access enables data retrieval in a predictable and efficient sequence, ensuring data is fetched from the memory and delivered in time. When a module issues a request, the requested data, already in the cache, is supplied with minimal latency, significantly enhancing data retrieval speed. The following section delves into the architecture and operation of the Time-Triggered Memory Access Controller (TTmaC), the core of the TTmaVTA, which orchestrates time synchronisation, dispatches requests, manages data responses into caches, and coordinates VTA module request handling.

Time-Triggered Memory Access Controller (TTmaC)

Memory controllers are essential for facilitating reliable communication between processors and system memory. They issue and manage commands for memory-related tasks, handle multiple memory access requests, manage memory bandwidth, and implement arbitration strategies [122] to prevent conflicts and maximise throughput [121]. Existing memory controllers also optimise access patterns by managing

memory banks and rows, aiming to reduce latency and improve memory efficiency. However, these controllers often encounter challenges such as resource contention [23], variable latency [24], and unpredictable memory access behaviour [25]. These issues can lead to underutilisation of memory bandwidth [26] and unfair access during concurrent DRAM access attempts [139], highlighting the need for custom controllers for specific applications that can interface effectively with existing memory controllers. Additionally, the dispatcher, a trigger component of time-triggered architectures, requires an application-tailored controller to manage its operations.

To address these needs, while considering the characteristics of VTA modules and their memory access patterns represented by the memory access trace analysis in Section 3.4, the TTmaVTA introduces a specialised controller called the Time-Triggered Memory Access Controller (TTmaC), depicted in the block diagram in Figure 3.12, to manage the TTmaVTA operations.

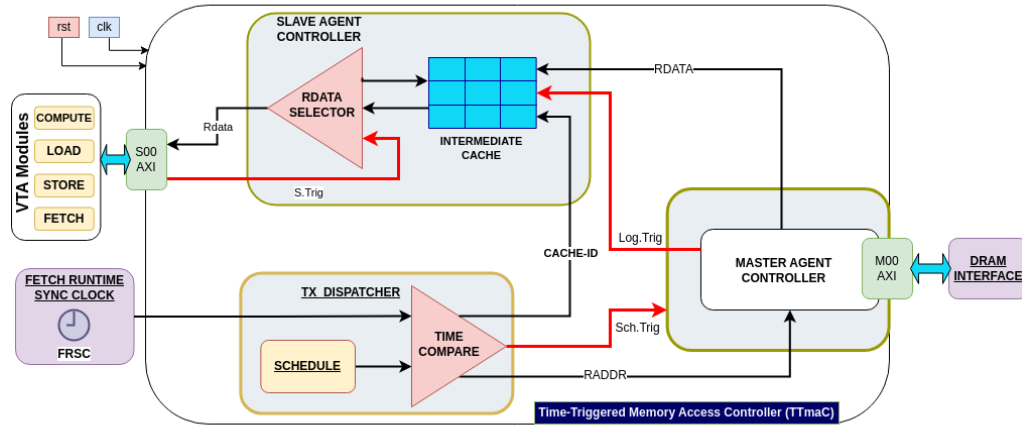


Figure 3.12: Time-Triggered Memory Access Controller (TTmaC) Block Diagram

The TTmaC is designed to interface with the VTA modules directly and operates at the top hardware level, orchestrating memory accesses based on a predefined schedule to ensure predictable neural network memory accesses while improving system memory performance. During execution, it assigns various neural network memory operations according to the scheduled times, effectively streamlining different access stream patterns. The schedule for VTA memory access generated in Section 3.5, based on the analysis of the memory access patterns from the traced access streams, is loaded to the PL at VTA initialisation during startup and stored within the TTmaC’s cache memory.

The Time-Triggered Memory Access Controller (TTmaC) is a custom-designed controller developed using Hardware Description Language (HDL) that performs preemptive data retrieval from memory based on a pre-computed schedule. The TTmaC, housed on the Programmable Logic (PL), serves as a bridge interface that connects the VTA modules to the Advanced eXtensible Interface (AXI) smart interconnect mapped to the slave high-performance computing (HPC) port of the Zynq UltraScale+ MPSoC PS, directly interacting with the DDR subsystem. Integrated into the TTmaVTA architecture on the FPGA, the TTmaC operates using pre-defined time slots and is synchronised by the global clock (FRSC). The TTmaC

manages requests from the VTA modules, interacting with the main memory by handling memory requests and responses based on the generated schedule. The TTmaC ensures time-based execution, such that memory accesses occur at precise, predetermined intervals. This approach ensures timely and orderly access to memory resources, optimising system performance.

The TTmaC utilises a concept of data request and retrieval from the main memory at predefined times, ensuring memory accesses occur within temporal constraints while providing temporal predictability. The TTmaC consists of four main components: the schedule dispatcher that manages the schedule entries and initiates memory access triggers based on predefined time slots, the Master Agent Controller that oversees interactions with the main memory, handling memory requests with AXI transactions and ensuring proper data management in the cache, the Slave Agent Controller that interfaces with the VTA modules and intermediate cache to retrieve the already stored data in the cache and pass the data to the requester module, while the Intermediate Cache allows for shared cache access between the Master Agent Controller and Slave Agent Controller, providing temporary storage to improve access times with reduced latency. The components that make up the TTmaC, as elaborated in the design of Figure 3.13, perform these operations:

- **Schedule Dispatcher:** The schedule dispatcher comprises two components: the schedule preload memory and the time comparator dispatch element. The 64-bit schedule created in Section 3.5 is loaded from a text file during VTA initialisation into a cache memory housed in the TTmaC, which stores the generated schedule entries. The first entry is automatically activated immediately after initialisation to invoke the comparator dispatch element, which begins checking the 28-bit time slice of the schedule against the global clock. The schedule dispatcher of the TTmaC operates based on the HDL (Hardware Description Language) state machine detailed in Figure 3.14.

Once the scheduled time entry of a memory operation matches the global clock (FRSC), a command is passed to enable the activation of the Master Agent Controller to retrieve data from the main memory. For each time instance of a schedule entry that matches the global clock, the corresponding address of that entry is used to identify the correct location of the data in the main memory for retrieval. The Cache ID is simultaneously passed to the shared intermediate cache to identify the slot for storing the retrieved data. The time comparator also serves as the schedule entry selector, reading each time instance of the schedule entries to activate the Master Agent Controller and transitioning to the subsequent entry after processing the previous one, continuing until all schedule entries are fulfilled.

- **Master Agent Controller:** The Master Agent Controller has a direct interface to the main memory and initiates a transaction with the main memory via an AXI interface when triggered by the schedule dispatcher when the scheduled time matches the global clock. It initiates the memory access request by sending a read/write valid signal to the AXI bus, along with the address of the requested data and the AXI burst transfer parameters (stride, x_size, and y_size).

Since this request is time-triggered, the response is received with minimal latency due to the absence of memory contentions or delays from concurrent

accesses. Once the AXI read/write transaction is completed, the data response is retrieved on a valid response signal and stored in the intermediate cache region specified by the Cache ID. The next transaction proceeds with the subsequent valid trigger signal from the schedule dispatcher.

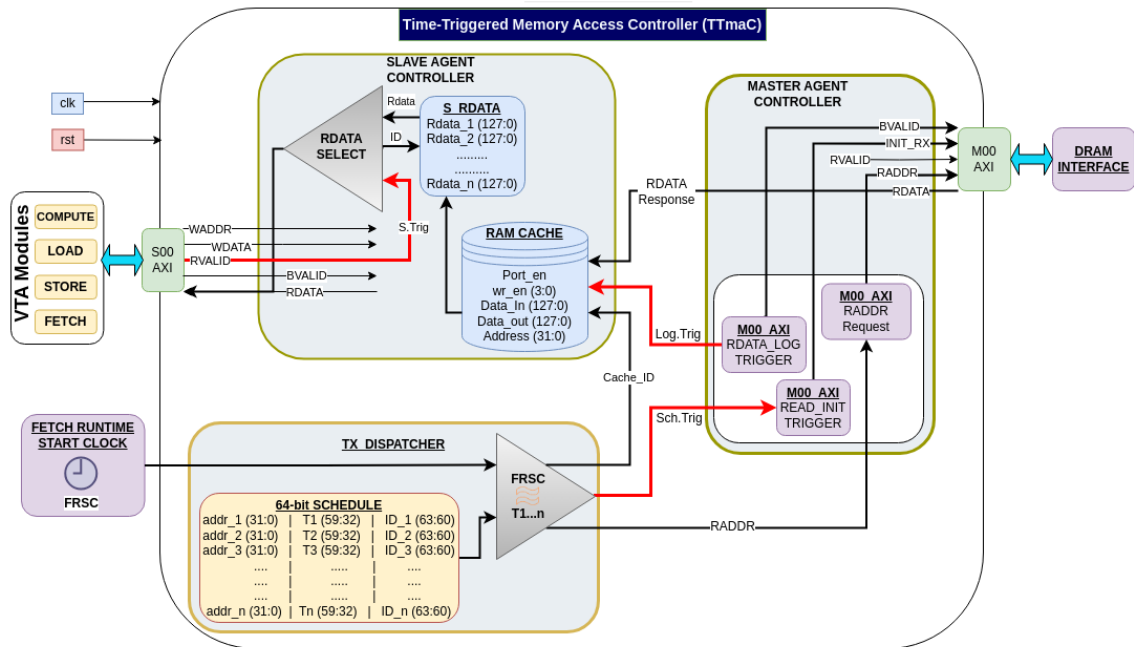


Figure 3.13: Time-Triggered Memory Access Controller (TTmaC)

- Slave Agent Controller:** The Slave Agent Controller interfaces with the VTA modules and waits for a valid request from each module. Once a valid request is made, it uses the module request address to look up the intermediate cache memory region for the appropriate Cache ID as assigned by the cache identifier. It then retrieves the data already cached by the Master Agent Controller and passes it to the requester VTA module via the Master AXI read data port interface. The Slave Agent Controller is thus activated when any of the VTA modules make an AXI read request, and the data retrieval is achieved by mapping the Slave Agent Controller’s registers to the memory regions of the intermediate cache based on the cache identifier.
- Intermediate Cache:** This cache serves as an intermediary memory between the Master Agent Controller and the Slave Agent Controller during each cycle interval. For example, in a cycle where the Master Agent Controller receives a trigger command from the schedule dispatcher at a schedule entry time match, it requests data from the memory, retrieves it, and activates the cache enable port to write the data to the allocated cache memory region. The Slave Agent Controller then accesses the same cache to retrieve the data when the VTA module makes a request. This cycle continues for each schedule entry until all neural network memory-related tasks are fully executed.

The state machine in Figure 3.14 is designed to manage the scheduling and execution of memory access operations within a time-triggered architecture.

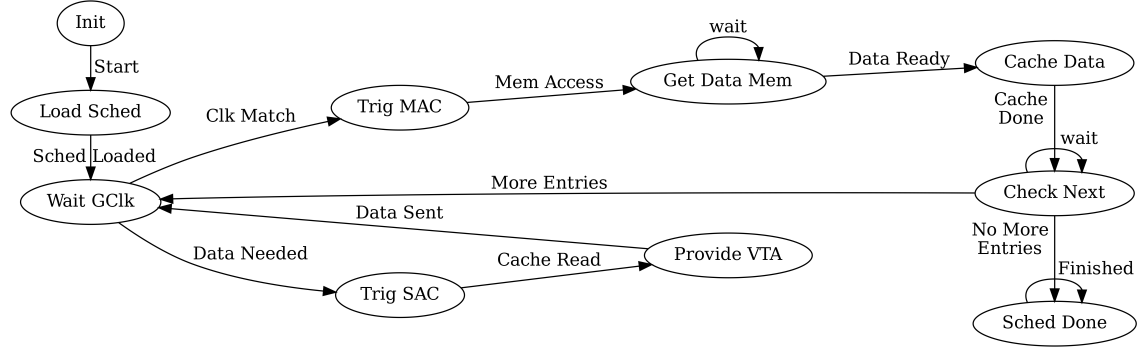


Figure 3.14: Schedule Dispatcher State Machine

Each state in the state machine represents a specific phase in the scheduling process, while the transitions between states ensure that memory operations are conducted in a deterministic and orderly manner, consistent with the time-triggered architecture. The sequence of operations are as follows:

- **Initialise (Init):** This state initialises the schedule dispatcher, setting up the necessary parameters and preparing the system to load the scheduling information.
- **Load Schedule (Load Sched):** After initialisation, the system transitions to the Load Schedule state, where the schedule data is retrieved and stored in memory. This schedule includes precomputed timings and addresses for memory operations, which are critical for maintaining precise timing and avoiding potential conflicts.
- **Wait for Global Clock Match (Wait GClk):** Following the scheduled load, the system enters the Wait for Global Clock Match state, where it remains idle until the global clock (FRSC) matches the time specified in the next schedule entry. This ensures that memory access operations are triggered at the exact scheduled times, maintaining temporal determinism.
- **Trigger Master Agent Controller (Trig MAC):** When the global clock matches a scheduled time, the state machine transitions to the Trigger Master Agent Controller state. In this state, the Master Agent Controller is activated to initiate the memory access, ensuring that data retrieval from the main memory occurs without delay or contention.
- **Retrieve Data from Main Memory (Get Data Mem):** In this state, the Master Agent Controller retrieves the necessary data from the main memory using the address specified in the schedule. The memory access is managed to adhere to timing constraints, and the retrieved data is then prepared for storage in the intermediate cache.
- **Store Data in Intermediate Cache (Cache Data):** Once data retrieval is complete, the state machine transitions to the Store Data in Intermediate Cache state. Here, the data is temporarily stored in the intermediate cache. The Cache ID, obtained from the schedule, determines the precise location within the cache, facilitating optimised data retrieval for subsequent accesses.

- **Check for Next Schedule Entry (Check Next):** After caching the data, the system checks for any remaining schedule entries in the Check for Next Schedule Entry state. If additional entries are present, the state machine loops back to Wait for Global Clock Match, ensuring the continuous execution of scheduled operations.
- **End of Schedule (Sched Done):** If no further schedule entries remain, the state machine transitions to the End of Schedule state, marking the completion of all scheduled tasks.
- **Trigger Slave Agent Controller (Trig SAC):** Upon detecting a data request from the VTA module, the state machine transitions to the Trigger Slave Agent Controller state, where the Slave Agent Controller is activated to access the intermediate cache and prepare the requested data.
- **Provide Data to VTA Module (Provide VTA):** In the Provide Data to VTA Module state, the data stored in the intermediate cache is delivered to the VTA module. This state ensures the efficient transfer of data to support ongoing operations in the VTA module.

Transitions between these states are driven by the global clock and the timing constraints specified in the schedule. The state machine efficiently manages the flow of data from the main memory to the cache and ultimately to the VTA module, ensuring that all operations are performed in a predictable and conflict-free manner. This approach enhances system determinism and optimises memory access and data retrieval processes, improving overall system performance.

The TTmaC design focuses on meeting time-triggered requirements while maintaining a simplified architecture. Given that the VTA already utilises multiple SRAMs, a more complex TTmaC design or a larger intermediate cache could result in insufficient SRAM availability, potentially causing synthesis issues due to routing congestion in such large designs [151] and non-synthesizable hardware solutions [32]. Cache size is a critical factor, as incorrect sizing can lead to challenges in hardware resource management. Therefore, the TTmaC design carefully considers hardware resource constraints when determining the appropriate cache size.

The utilisation of maximum latencies of each memory access for schedule generation introduces overhead in execution time due to factors such as conservative timing margins, increased buffer allocations, and potential underutilisation of system resources. These factors lead to inefficiencies that require further attention. To mitigate the impact of these overheads, the schedules are tightened, and advanced memory latency optimisation techniques are implemented to reduce the argument overheads in Chapter 4.

Chapter 4

Optimisation of Neural Network Memory Accesses

The design of processor architectures for specialised application domains, such as safety-critical systems [65] [152] [153], has experienced significant growth due to the rapid advancements in neural networks. These developments have driven the architectural design of dedicated platforms, such as Graphics Processing Units (GPUs), Edge Artificial Intelligence (EdgeAI), and Field-Programmable Gate Arrays (FPGAs) for neural network acceleration [154] [155] [156], as detailed in Section 2.2. These accelerators handle the complex computations associated with neural networks, such as CNNs discussed in Section 2.1, which demand more processing capacity from existing platform architectures, but are often difficult to meet. Although GPUs are powerful, they are expensive and constrained by power consumption in certain application areas. FPGAs, on the other hand, have garnered increased attention due to their portability, flexibility, and energy efficiency; however, they pose challenges in design and optimisation for resource-intensive applications. One significant issue is the limited computing resources of FPGAs, making them susceptible to strain from applications with heavy computational demands and complex memory access patterns, such as neural networks. The available memory bandwidth on an FPGA presents a significant challenge, given the potential mismatch between the high throughput requirements of neural network computations and the memory bandwidth provided by the FPGA. This chapter introduces memory optimisation techniques to address the memory access inefficiencies in baseline VTA architectures.

4.1 Memory Optimisation Techniques and Strategies

As reviewed in Section 2.7, many existing approaches have not fully exploited the memory access potential of FPGAs for neural network deployment, particularly in addressing memory-related issues exacerbated by the increasing complexity and scalability of deep learning applications. Zhang et al. [154] observed that the computational throughput of current accelerators may not align well with the memory bandwidth provided by FPGA platforms, especially as deep learning models grow in complexity and scale. While the modified VTA, referred to as TTmaVTA, discussed in Section 3.6, has demonstrated improved performance in terms of predictability

and timing guarantees, the full potential of the accelerator design space and memory mappings has not been fully explored. Although the time-triggered approach addresses the problem of predictability, it introduces time overhead due to its scheduling approach, which deploys conservative timing margins where all timing is based on the best-case memory access time, potentially leading to underutilisation of system resources. This execution latency is further exacerbated by memory-related issues, either to the underutilisation of logic resources or the inefficient use of memory bandwidth on FPGA deployments by existing accelerators. Therefore, relying solely on the time-triggered approaches proposed in Section 3.6 for this dissertation may not achieve optimal performance on a resource-limited platform, such as FPGAs.

To improve upon the previous design in Chapter 3, address these memory-related issues, and achieve optimal performance, this chapter employs an analytical memory co-design approach based on a quantitative analysis of VTA computational throughput, leveraging the memory traces discussed in Section 3.3. The focus is enhancing memory bandwidth for neural network accelerators, such as the VTA deploying neural network models like CNNs, by employing optimisation techniques. This dissertation, therefore, proposes optimising memory access performance through various strategies, including prefetching and caching, thereby enhancing memory performance during FPGA memory accesses. The insights gained from the memory access analysis in Section 3.5 allow for proposing optimisation techniques aimed at minimising idle memory phases and mitigating dependency bottlenecks.

The analysis of memory access traces for neural networks on the VTA, as conducted in Section 3.4 of Chapter 3, provides deep insights into bottlenecks such as memory access overlaps, idle phases, and dependency waiting. The application of network flow graphs further aids in visualising and analysing these issues. The use of time-triggered scheduling in Section 3.5 offers a powerful optimisation technique that aligns memory operations within predefined time slots, eliminating overlaps and improving efficiency. By generating optimised schedules, neural networks can achieve better memory utilisation with lower latency, critical in performance-sensitive applications such as AI accelerators.

However, the latency formula of Equation 3.7 does not account for delays caused by memory operations that depend on other operations, which can lead to memory bus contention due to unfulfilled dependencies. This behaviour can cause avoidable memory access delays, leading to memory idleness, where other memory operations with non-dependent or completed dependent tasks could be leveraged to maximise resources and improve execution time. Additionally, Equation 3.7 does not consider factors such as memory access contention due to concurrent memory access attempts, nor does it address the frequency of memory accesses to the same address locations, requesting data at short consecutive intervals. The proposed memory optimisation techniques involve the implementation of **prefetching** to mitigate dependency-related bottlenecks and **data caching** to improve the efficiency of frequently accessed memory operations. These methods aim to address the memory access limitations encountered in the Versatile Tensor Accelerator (VTA). Prefetching is used to load data ahead of its expected request by reordering execution, reducing the latency caused by memory access dependencies. Caching, on the other hand, stores frequently accessed data closer to the computation units, minimising redundant memory fetches. Together, these techniques improve memory throughput

and computational efficiency, particularly in handling the memory-bound operations typical in VTA workloads.

4.2 VTA with Optimised Time-Triggered Memory Access (OpTTmaVTA)

This section introduces two memory optimisation strategies: idle memory phase utilisation with data prefetching and caching of frequently accessed data with repetitive memory access. Each strategy is accompanied by algorithms and mathematical models to quantify latency reductions and performance improvements.

4.2.1 Idle Memory Phase Utilisation with Data Prefetching in Memory Access

Idle memory phases occur when memory operations are stalled, particularly due to dependency-related delays. A prefetching mechanism is introduced to exploit these idle phases by reordering execution and optimising the timing of operations. Idle time is minimised by allowing memory accesses with no dependencies, or those whose dependencies have already been resolved, to execute earlier if the available window permits. Specifically, suppose the next memory operation in the sequence has unfulfilled dependencies, in that case, a subsequent operation can be prefetched and executed instead of allowing the bus to remain idle, reducing the total idle time.

The total idle time T_{idle} in the system can be computed as:

$$T_{\text{idle}} = \sum_{i=1}^n (t_{i+1}^{\text{start}} - t_i^{\text{finish}}) \cdot \mathbf{1}_{(t_{i+1}^{\text{start}} > t_i^{\text{finish}})} \quad (4.1)$$

Where n is the total number of operations, and $\mathbf{1}$ is an indicator function that evaluates to 1 if the condition is true, and 0 otherwise.

This approach is enhanced by incorporating prefetching as follows:

$$T_{\text{idle}} = \sum_{i=1}^n (t_{i+1}^{\text{start}} - t_i^{\text{finish}}) \cdot \mathbf{1}_{(t_{i+1}^{\text{start}} > t_i^{\text{finish}})} - T_{\text{prefetchedOP}} \quad (4.2)$$

As shown in Equation 4.2, the time spent on prefetching operations $T_{\text{prefetchedOP}}$ is deducted from the total idle time, thereby reducing the overall idle time. However, incorporating the prefetched operations into Equation 4.2 requires accounting for the total time saved by executing prefetched operations during what would otherwise be idle periods. The aim is to quantify the impact of prefetching by measuring how much idle time is reduced when a memory operation that would otherwise be stalled is replaced by a prefetched operation. The following factors are considered to compute the time saved by prefetching operations:

- The number of prefetched operations executed during otherwise idle time.
- The time duration of these prefetched operations.
- The original expected idle time if no prefetching had been performed.

Two terms are defined from these factors: *Prefetched Operations Time* and *Idle Time Reduction due to Prefetching*, which are described as follows:

- (A) **Prefetched Operations Time:** The total time taken by prefetched operations executed during idle phases.
- (B) **Idle Time Reduction due to Prefetching:** The reduction in idle time is achieved by executing prefetched operations.

The reduction in idle time due to prefetching can be expressed as:

$$T_{\text{prefetchedOP}} = \sum_{j \in \text{Prefetched}} (t_j^{\text{finish}} - t_j^{\text{start}}) \quad (4.3)$$

Where:

- $T_{\text{prefetchedOP}}$ is the total time taken by prefetched operations.
- t_j^{start} and t_j^{finish} are the start and finish times of the prefetched operation j .

Thus, the extended formula for total idle time T_{idle} in the system, accounting for prefetching optimisation, is computed as:

$$T_{\text{idle}} = \sum_{i=1}^n (t_{i+1}^{\text{start}} - t_i^{\text{finish}}) \cdot \mathbf{1}_{(t_{i+1}^{\text{start}} > t_i^{\text{finish}})} - \sum_{j \in \text{Prefetched}} (t_j^{\text{finish}} - t_j^{\text{start}}) \quad (4.4)$$

Where:

- t_i^{start} and t_i^{finish} are the start and finish times of operation i .
- $\mathbf{1}_{(t_{i+1}^{\text{start}} > t_i^{\text{finish}})}$ is an indicator function that is 1 if the next operation starts after the current one ends (indicating idle time), and 0 otherwise.
- t_j^{start} and t_j^{finish} are the start and finish times of the prefetched operation j .
- $\sum_{j \in \text{Prefetched}} (t_j^{\text{finish}} - t_j^{\text{start}})$ represents the total time taken by prefetched operations during idle periods.

Equation 4.4 demonstrates that the prefetching technique helps to reduce the idle phases by $T_{\text{prefetchedOP}}$, achieved by executing independent operations out of order when dependency-related operations would have otherwise stalled the next sequential operation. This formula ensures that the total idle time is reduced by the time spent executing prefetched operations, thereby minimising idle phases by incorporating Algorithm 3.

This algorithm is employed to optimise the order and timing of memory accesses, minimising idle memory phases, which occur when operations are stalled due to dependency-related bottlenecks, by reordering executions and scheduling memory operations efficiently. It begins by sorting the list of memory accesses by their sequence number and timestamp and identifying those ready for execution (i.e. those that have no dependencies or have already resolved dependencies), placing them in a ready queue. It then iterates through each pair of consecutive accesses, comparing the start time of the following access with the finish time of the current one.

When an idle phase is predicted or detected—indicated by a timing gap between the end time of the current operation and the start time of the next—the algorithm checks if dependencies of the subsequent operation are unmet (i.e., when the next scheduled operation has unfulfilled dependencies that would halt execution). If so, it

Algorithm 3 Idle Phase Minimisation with Prefetching in Memory Accesses

```

1: Input: List of Memory Accesses  $\{A_1, A_2, \dots, A_n\}$ 
2: Output: Optimised Access Schedule
3: Sort memory accesses by sequence number and timestamp
4: Initialise  $schedule \leftarrow []$ 
5: Initialise  $ready\_queue \leftarrow []$  ▷ Queue to store ready operations
6: for  $i = 1$  to  $n$  do
7:   Check Dependencies for  $A_i$ 
8:   if All dependencies of  $A_i$  are fulfilled then
9:     Add  $A_i$  to  $ready\_queue$ 
10:  end if
11: end for
12: Sort  $ready\_queue$  by earliest time and priority
13: for  $i = 1$  to  $n - 1$  do
14:   if  $t_{i+1}^{start} > t_i^{finish}$  then ▷ Idle phase detected
15:     Check Dependencies for  $A_{i+1}$ 
16:     if Dependencies of  $A_{i+1}$  are unfulfilled then
17:       Select Next Prefetchable Operation from  $ready\_queue$ 
18:       Execute Prefetched Operation before  $A_{i+1}$ 
19:     else
20:       Continue Executing  $A_{i+1}$ 
21:     end if
22:   else
23:     Continue with normal sequence
24:   end if
25:   Append  $A_i$  to  $schedule$ 
26: end for
27: Return Optimised Schedule

```

selects a prefetched operation from the ready queue, adjusts its start time to fit into the idle window, and schedules it for execution during this anticipated idle window. If no idle phase is detected, no adjustment is necessary, and the usual sequence continues. This process ensures continuous memory bus utilisation, minimising idle periods.

This prefetching mechanism, as detailed in the pseudocode of Algorithm 11 in Appendix, can be integrated into the time-triggered scheduling algorithm of Section 3.5 to generate a more optimised schedule, which in turn improves memory throughput and reduces inference time, compensating for the overheads introduced in Chapter 3. The final output is an optimised access schedule that mitigates avoidable delays and aligns operations to improve memory system efficiency.

4.2.2 Cache-Based Optimisation of Repetitive Memory Access

In neural network implementations on hardware, optimising memory access is critical to achieving high performance. In this section, the focus is on optimising memory operations through caching frequently accessed data. By storing repetitive write-read operations in small caches (e.g., BRAM), the number of accesses to main memory is reduced, alleviating the burden of frequent memory access on the memory bus. This section details the identification of cacheable operations, cache size and bandwidth calculations, and the associated benefits of caching in terms of reduced memory access, reduced execution time, and improved memory throughput.

4.2.2.1 Caching Frequently Accessed Data

The caching optimisation begins with the identification of frequently accessed memory addresses. Neural network operations typically involve a significant number of repetitive accesses to specific memory locations, particularly during input and weight loading, activation readouts, or intermediate data storage and retrieval. Caching these repetitive accesses can significantly reduce the load on the main memory.

Let the memory access trace be denoted by the sequence of memory operations:

$$\{(\text{Address}_i, \text{Op}_i, t_i)\}_{i=1}^N$$

Where:

- Address_i represents the memory address.
- Op_i represents the type of operation (either read or write).
- t_i is the time step of operation i .
- **Identifying Cacheable Operations:** The process starts with identifying frequently accessed addresses and counting their occurrences. Let $\text{Frequency}(\text{Address}_i)$ denote the number of times address Address_i is accessed within the trace. If the frequency exceeds a predefined threshold between consecutive write operations of the same address, the address is considered cacheable:

$$\mathcal{A}_{\text{cacheable}} = \{\text{Address}_i \mid \text{Frequency}(\text{Address}_i) \geq \text{Threshold}\}$$

Addresses in the set $\mathcal{A}_{\text{cacheable}}$ are assigned a cache block from the cache memory $\mathcal{C}$, where $\mathcal{A}_{\text{cacheable}}$ is the set of cacheable memory addresses. The threshold is determined by computing Equation 4.12.

- **Cache Management and Write-Read Pairing:** Once cacheable addresses are identified, cache memory blocks (e.g., BRAM) are assigned to each unique cacheable address to store the data. For every write operation to a cacheable address, the corresponding data is stored in the cache. The cache is organised by assigning identifiers (*IDs*) to the memory blocks used for caching *write-read pairs*. Whenever a memory operation matches an address in $\mathcal{A}_{\text{cacheable}}$, it redirects to the cache.

The mapping is represented as:

$$\text{ID}(\text{Address}) = \mathcal{C}[\text{Address}]$$

Where:

- $\mathcal{C}$ represents the set of cache blocks.
- $\text{ID}(\text{Address})$ is the cache ID assigned to a specific memory address.

Each write operation updates the corresponding cache block, and each subsequent read operation accessing the same address is redirected to the cache instead of accessing the main memory.

Let $\text{Cache}_{\text{write}}(t_i)$ represent a write operation to the cache and $\text{Cache}_{\text{read}}(t_j)$ a read operation.

$$\text{Cache}_{\text{write}}(t_i) \rightarrow \text{invalidate}(t_{i-1})$$

If a new write operation occurs, it invalidates the old cache entry and replaces it with the new data, ensuring data consistency for all cacheable operations.

4.2.2.2 Cache Size and Bandwidth Allocation

The cache size (in terms of depth and width) is determined by the data size of the operations and the characteristics of its AXI burst transfer rates (Stride, x_{size} , y_{size}) as part of the traced data.

The cache depth for a given memory block is determined by:

$$\text{Cache Depth} = \frac{\text{Data Size}}{\text{Width}} \times \frac{y_{\text{size}}}{\text{Stride}} \quad (4.5)$$

The bandwidth required for the cache can be calculated as:

$$\text{Bandwidth} = x_{\text{size}} \times y_{\text{size}} \quad (4.6)$$

Then, the base address of the next cacheable pair is determined by:

$$\text{Next Cache Base Address} = \text{Previous Base Address} + \text{Cache Depth} \quad (4.7)$$

$$\text{Next Cache Base Address} = \text{Previous Base Address} + \left(\frac{\text{Data Size}}{\text{Width}} \times \frac{y_{\text{size}}}{\text{Stride}} \right) \quad (4.8)$$

The calculation of the cache size and bandwidth ensures that the cache can handle the necessary data load, reducing the number of main memory accesses for frequently accessed addresses.

4.2.2.3 Total Time Saved by Caching

Caching frequently accessed memory addresses allows for faster access to data in the cache and reduces the total number of main memory accesses, thereby saving memory access time and decreasing the overall neural network execution time. The total time saved by caching, denoted by T_{cacheOP} , is computed based on the number of cache hits and the time required to access the main memory versus the time to access the cache and calculated as:

$$T_{\text{cacheOP}} = N_{\text{hits}} \times (T_{\text{mem}} - T_{\text{cache}}) \quad (4.9)$$

Where:

- T_{mem} : Time required for a main memory access.
- T_{cache} : Time required for a cache access.
- N_{hits} : Number of cache hits (i.e., read operations redirected to cache).

This formula in Equation 4.9 provides a deeper reflection of the total time saved after implementing cache optimisation. The next step involves managing the cache effectively to capitalise on these potential time savings. Algorithm 4 outlines the process of managing and optimising cache usage to enhance memory access performance by efficiently caching and reusing frequently accessed data.

Algorithm 4 Cache Management and Optimisation

```

1: procedure CACHEMANAGEMENTOPTIMISATION(Memory Access Trace, Cache Size  $S$ )
2:   Initialise cache  $\mathcal{C}$  with size  $S$ 
3:   Initialise cacheable set  $\mathcal{A}_{\text{cacheable}} \leftarrow \emptyset$ 
4:   for each memory access ( $\text{Address}_i, \text{Op}_i, t_i$ ) do
5:     Calculate Frequency( $\text{Address}_i$ )
6:     if Frequency( $\text{Address}_i$ )  $\geq$  Threshold then
7:       Add  $\text{Address}_i$  to  $\mathcal{A}_{\text{cacheable}}$ 
8:       Assign cache block  $\mathcal{C}[\text{Address}_i]$ 
9:     end if
10:  end for
11:  for each memory access ( $\text{Address}_i, \text{Op}_i, t_i$ ) do
12:    if  $\text{Address}_i$  is in  $\mathcal{A}_{\text{cacheable}}$  then
13:      if  $\text{Op}_i$  is a write then
14:        Write data to  $\mathcal{C}[\text{Address}_i]$ 
15:        Invalidate previous cached data
16:      else if  $\text{Op}_i$  is a read then
17:        Redirect read to cache  $\mathcal{C}[\text{Address}_i]$ 
18:      end if
19:    end if
20:  end for
21:  Calculate time savings  $T_{\text{cacheOP}}$ 
22:  Update idle time  $T_{\text{idle}} \leftarrow T_{\text{idle}} - T_{\text{cacheOP}}$ 
23:  return Optimised schedule and cache configuration
24: end procedure

```

The algorithm presented in Algorithm 4 focuses on optimising memory access by effectively managing cache resources to provide an optimised schedule. This caching-based optimisation technique is particularly beneficial for improving memory access efficiency in neural networks, where memory bandwidth often presents a significant bottleneck.

The algorithm begins by taking as input the memory access trace from Section 3.3, which consists of a sequence of memory addresses, operations (read or write), and corresponding timestamps. Additionally, the algorithm requires the size of the cache, denoted as S , obtained from Equations 4.5 and 4.6. The cache structure, $\mathcal{C}$, is then initialised with the specified size S , ready to temporarily store frequently accessed memory addresses. An empty set, $\mathcal{A}_{\text{cacheable}}$, is also initialised to hold memory addresses that meet the criteria for caching based on their access frequency, given by a threshold whose computation is given by Equation 4.12.

The next phase of the algorithm involves identifying cacheable addresses. It iterates over each memory access in the trace and calculates the frequency of each memory address. Suppose the frequency of a particular address meets or exceeds a predefined threshold, indicating that it is accessed frequently. In that case, the address is added to the cacheable set $\mathcal{A}_{\text{cacheable}}$. Subsequently, the memory address is assigned a block in the cache $\mathcal{C}$, preparing it for quicker access in subsequent operations.

After identifying cacheable addresses, the algorithm manages cache operations. For each memory access, the algorithm checks whether the address is present in the cacheable set $\mathcal{A}_{\text{cacheable}}$. If the operation is a write, the data is written to the corresponding cache block $\mathcal{C}[\text{Address}_i]$, and any previously cached data for that address is invalidated to maintain data consistency. For read operations, if the address is found in the cache, the read is redirected to the cached data, thus alleviating the

burden on the memory bus and reducing the access time compared to fetching the data from the main memory.

In the final stage of the algorithm, the total time savings, T_{cacheOP} , achieved by using the cache is calculated. This time saving reflects the reduced access times for addresses served from the cache rather than from the main memory. The idle time in the system, T_{idle} , is then updated by deducting the calculated time savings, indicating the improvement in system efficiency due to the memory access caching.

The proposed method is crucial for enhancing the performance of neural network deployment in hardware, as it reduces main memory access latency, improves memory access efficiency, and increases system throughput. By efficiently managing cache blocks and ensuring data consistency, this algorithm plays a key role in optimising memory access and patterns, particularly in environments with limited memory bandwidth, such as FPGAs.

The algorithm concludes by returning the optimised memory access schedule along with the final cache configuration for integration into the TTmaVTA. This optimised schedule ensures that frequently accessed memory addresses are efficiently served from the cache, thereby minimising idle time and improving overall memory access performance. The detailed steps and calculations involved in this process provide a comprehensive approach to optimisation of memory access, demonstrating cache management's effectiveness in enhancing system performance.

4.2.3 Reduction in Idle Time

Both approaches—prefetching, discussed in Section 4.2.1, and caching, detailed in Section 4.2.2—help to reduce idle phase durations during memory access. Idle memory phases occur when operations are stalled due to dependencies or memory contention. By prefetching data during idle phases and caching frequently accessed data, the waiting time for memory accesses is reduced, thereby minimising idle time.

The formula in Equation 4.11 accurately reflects the system's idle periods after implementing cache optimisation. The total idle time from Equation 4.2 is recalculated to account for caching:

$$T_{\text{idle}} = \sum_{i=1}^n (t_{i+1}^{\text{start}} - t_i^{\text{end}}) \cdot \mathbf{1}_{(t_{i+1}^{\text{start}} > t_i^{\text{end}})} - T_{\text{prefetchedOP}} - T_{\text{cacheOP}} \quad (4.10)$$

Equation 4.10 can be extended by incorporating the formulas from Equations 4.3 and 4.9, resulting in:

$$T_{\text{idle}} = \sum_{i=1}^n (t_{i+1}^{\text{start}} - t_i^{\text{end}}) \cdot \mathbf{1}_{(t_{i+1}^{\text{start}} > t_i^{\text{end}})} - \sum_{j \in \text{Prefetched}} (t_j^{\text{end}} - t_j^{\text{start}}) - (N_{\text{hits}} \times (T_{\text{mem}} - T_{\text{cache}})) \quad (4.11)$$

This time-saving directly translates into a reduction in total memory access latency, with the combined optimisation approaches leading to improved system memory throughput.

4.3 Software Simulation of Versatile Tensor Accelerator (VTA) Operations: Functional Modelling for Optimisation

While the hardware-based approach to data tracing explored in Section 3.3 is effective, providing more accurate trace data, hardware benchmarking of neural networks on edge FPGAs with large expected trace data requires a significant cache size, which is resource-intensive [157]. Such resources are often unavailable in small- to mid-sized FPGAs, leading to repeated adaptation processes that are cumbersome and time-consuming [138] and necessitate repeated and extensive logic synthesis, which is costly [32]. Conversely, the software approach reduces the frequency of logic synthesis and provides approximate estimates through simulations. However, these simulations lack the ability to capture actual performance metrics [32]. Therefore, the software approach is primarily employed for simulating the behaviour of the optimisation approaches discussed in Section 4.2 to quantify and evaluate the applied optimisation techniques. [150] propose a VTA simulator that closely mimics the entire operation and deployment performance of the VTA, enabling close evaluation of various optimisation strategies. This simulator is leveraged in this dissertation to simulate the optimisation strategies for deploying the VTA. While section 4.3.1 describes the work of [150], the output from the simulator is further extended in the section 4.3.2 for analysis to enhance memory performance. This is achieved by optimising idle memory access periods, prefetching data when feasible, and caching frequently accessed write operations for subsequent reads.

4.3.1 VTA Software Simulator: An Architectural and Functional Review

The VTA software simulator, developed by [150], is a lightweight C++ tool designed to emulate the operations of tensors and instructions within the VTA architecture. This simulator provides the flexibility needed to analyse, modify, and optimise the behaviour of VTA, ultimately producing structured, time-sensitive data for the entire spectrum of VTA operations.

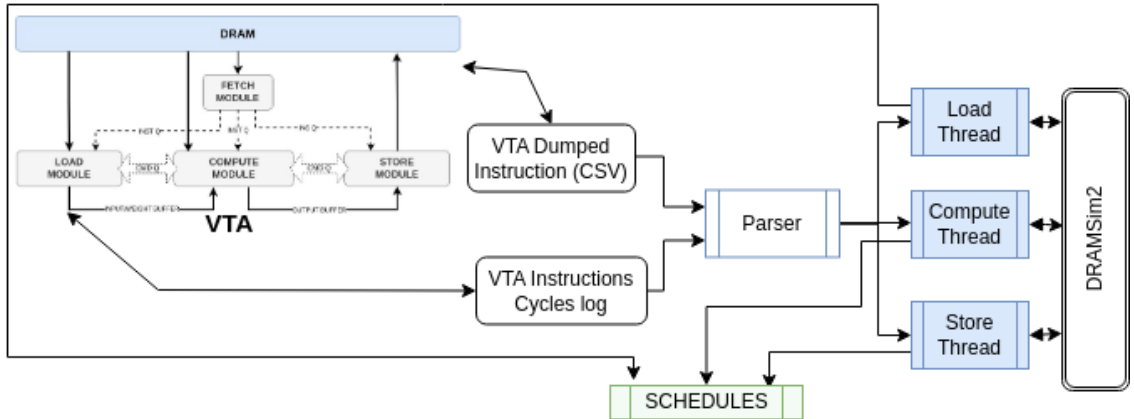


Figure 4.1: Versatile Tensor Accelerator (VTA) Simulator

The structure of the VTA software simulator, shown in Figure 4.1, includes com-

ponents such as a parser, DRAM Simulator (DRAMSim2), and module threads. The input for the simulator is derived from statistical data collected at runtime and from the simulation of the standard VTA operations. The statistical data necessary for the software simulator is obtained during the pre-simulation phase, which involves collecting runtime data and profiling VTA performance using a general-purpose simulation tool. These reorganised statistics serve as the foundation for generating the simulator’s input, enabling the examination of various configurations and assessing their impact on performance and resource utilisation.

4.3.1.1 Pre-Simulation Phase: Acquisition of VTA Runtime Statistical Data for Simulator Input Generation

Comprehensive data collection and organisation are essential to providing accurate inputs for the VTA simulation process. This step involves gathering detailed runtime metrics, such as execution time, memory access patterns, and data throughput, from various network architectures. These metrics are crucial for constructing a realistic simulator input that accurately reflects the neural network’s behaviour and performance on the target hardware. By organising these statistics, potential bottlenecks within the VTA can be identified, allowing for targeted optimisations and enhancements. This method ensures that the simulator accurately models the operational environment, providing valuable insights into the performance characteristics of the neural network and guiding the development of more efficient acceleration and optimisation strategies. Performance data is collected from executing a deep learning model, such as ResNet-18, chosen to experiment with the simulator and then demonstrate, implement, and evaluate the memory access optimisation techniques proposed in this dissertation. The ResNet-18 [158] model architecture is widely recognised for its effectiveness in image classification tasks. According to [150], the pre-simulation phase of the simulator design process focuses on two critical aspects: acquiring input data and profiling the VTA’s instruction cycle performance.

- **Input Data Acquisition:** The data collection process begins with extracting and recording instructions and operational details of the executed model (e.g., ResNet-18 model) during inference to gather performance data from the conventional VTA and convert them into structured data. The operational data of the VTA is collected during runtime by introducing a debug flag in the TVM compiler’s codebase within its inference pipeline. This flag captures VTA performance analytics, logging information such as operations (instructions), operation types (read or write), layers, dependencies, addresses, and other related details into a text file. The collected data includes unstructured details of the instructions and operations. A specialised software tool converts this text file into a format compatible with the VTA Software Simulator. However, these unstructured data cannot accurately represent the sequence and timing of execution; thus, a software simulation is required to process this information, determining the precise order and timing of operations. An input adaptive module of the simulator then converts this text file into a format compatible with the VTA software simulator. The formatted data serves as the simulator’s input, using queuing mechanisms managed by each module as threads to order the instruction processes according to their dependencies. Acquiring VTA instruction cycle performance data is essential for comprehen-

sive analysis, as the timing data for each instruction's execution period is not provided in the debugged information.

- **VTA Instruction Cycles Performance Profiling:** The initial data collected lacks the necessary timing or cycle information required for a thorough analysis. Therefore, employing Xilinx's Behavioural Simulation toolchain is essential for logging instruction cycle counts across various VTA modules. Since a comprehensive performance analysis of the VTA depends on both timing and cycle statistics, the Xilinx simulation toolflow, through its Behavioural Simulator, is used to capture VTA instruction cycle counts. The pseudocode in Algorithm 5, adapted from [150], simulates the operational data for the entire VTA process based on the collected statistical input data.

Algorithm 5 Simulation Logic

```

1: Input: "input_file.csv"
2: procedure MODULES_THREADS(modules, dependencies)
3:   Open file: schedule.txt
4:   Initialise modules and parse input: input_file.csv
5:   for each inst in inst_queue do
6:     inst.start_time  $\leftarrow$  0
7:     inst.finish_time  $\leftarrow$  0
8:     if inst.pop  $\neq$  False then
9:       Wait until pop_queue is not empty
10:      inst.start_time  $\leftarrow$  pop_queue.read()
11:    else
12:      prev_inst  $\leftarrow$  previous instruction in inst_queue
13:      inst.start_time  $\leftarrow$  prev_inst.finish_time
14:    end if
15:    if inst.type  $\neq$  0x00000000 and memory_controller is busy then
16:      Wait until memory_controller is free
17:      inst.start_time  $\leftarrow$  memory_controller.free_time()
18:      mem_cycles  $\leftarrow$  DRAM(R/W, offset, size={y, x, stride})
19:      inst.finish_time  $\leftarrow$  inst.start_time + mem_cycles
20:    else
21:      inst.finish_time  $\leftarrow$  inst.start_time + inst.log_cycles
22:    end if
23:    if inst.push  $\neq$  False then
24:      push_queue.write(inst.finish_time)
25:    end if
26:  end for
27:  Close file: schedule.txt
28: end procedure

```

The simulator processes the raw input data file, executing various instructions to produce valid operational data supporting further VTA operations analysis. The simulator processes and logs essential parameters needed for further analysis, including layer, instruction names, instruction start times, relative start and finish times, and dependencies, represented as *inst.layer*, *inst.name*, *inst.start_time*, *rel_start_time*, *rel_finish_time*, and *inst.dep*.

4.3.1.2 VTA Simulator Component Architecture

To effectively simulate the operational data of the entire VTA process, several key components are integrated within the VTA Software Simulator, as depicted in Figure 4.1. These components collaborate to emulate the timing, memory access, and instruction flow characteristic of the hardware version of the VTA. By leveraging simulation, the collected statistical input data is processed to replicate the hardware functionality, facilitating detailed performance analysis and optimisation. The primary components of the VTA Software Simulator are as follows:

- **Parser:** This component is responsible for syntactically analysing the input data and preparing them for the thread modules. The parser then interprets the input VTA performance metrics from the data, extracts the logged non-memory instruction cycle counts, and organises the instructions of each module into separate vector queues to aid in simulating the VTA functionality, thereby emulating the VTA Fetch module’s operations.
- **DRAM Simulator:** The timing statistics gathered in Section 4.3.1.1 are insufficient on their own, as they do not provide a coherent representation of VTA memory access timing. To address the complexities in the dynamics of simulating memory timings, the VTA simulator integrates DRAM simulation functionality, such as DRAMSim2. This integration provides a realistic representation of memory access times, improving the approximation of memory access timing within the VTA. It also allows for adapting timing information in conjunction with VTA instructions, their order, and cycle counts for a more in-depth analysis. DRAM simulators are essential for modelling the timing characteristics and latency of physical DRAM, offering a robust platform for comprehensive testing of memory-related functionalities.

Utilising DRAMSim2 [159], a cycle-accurate memory system simulator, this module models the timing characteristics and latency associated with physical DRAM. Configured to match the specifications of the FPGA, DRAMSim2 provides precise real-time memory estimates, facilitating accurate memory modelling within the system simulation. Unlike trace files or full-system simulators, which often lack an accurate model of the main memory system [160], DRAMSim2 is seamlessly incorporated into the C++ simulation framework of the VTA simulator, ensuring precise simulation and accurate memory estimates [159]. In this VTA simulation setup, DRAMSim2 functions as a dedicated main memory simulator, with device initialisation files specifically configured for Micron DDR2/3 devices and tailored configuration files that align with the FPGA hardware specifications, including memory size, data width, clock frequency, and period, ensuring precise memory estimations [161].

- **Module Threads:** This component represents the VTA modules, comprising Load, Compute, and Store threads. These threads retrieve instructions, analyse specific instruction patterns, and initiate memory transactions via the integrated DRAMSim2. These threads mirror the parallelism inherent in the VTA architecture across its modules. Instructions are fetched from the vector queues and may push or inherit dependencies. If a dependency is detected, the thread pauses execution and enters a queue until the dependency is resolved, at which point the queued instruction is executed. Upon completion

of all instruction execution, each thread generates a schedule representing the sequence of VTA instructions for further processing.

The entire simulation process integrates various components of the simulator alongside the parser. The file containing recorded and structured data from the VTA's runtime operations, including instruction and operation cycles, serves as input for the simulator. The integration of DRAMSim2 provides accuracy and flexibility in simulating the memory timing behaviour of the VTA under different scenarios and workloads, enabling valuable insights into the memory system's performance and identifying potential bottlenecks that can assist in the future optimisation of system design. The simulator's module threads utilise queuing mechanisms to sequence processes with dependencies, ultimately generating a comprehensive time-based schedule using software scheduling logic, suitable for future deployment in a time-triggered hardware setup.

4.3.2 Extension of the VTA Simulator for Enhanced Memory Handling and Access Optimisation

The VTA simulator is highly adaptable and reconfigurable. The limitations inherent in the conventional VTA remain observable in the VTA simulator platform, as the simulator directly reflects the VTA on hardware. To leverage the VTA simulator's flexibility and adaptability, its parameter capturing and logging capabilities have been enhanced. The simulator is adapted to generate time-based operational data for the entire VTA process during simulation, facilitating in-depth analysis, modification, and optimisation of VTA behaviour.

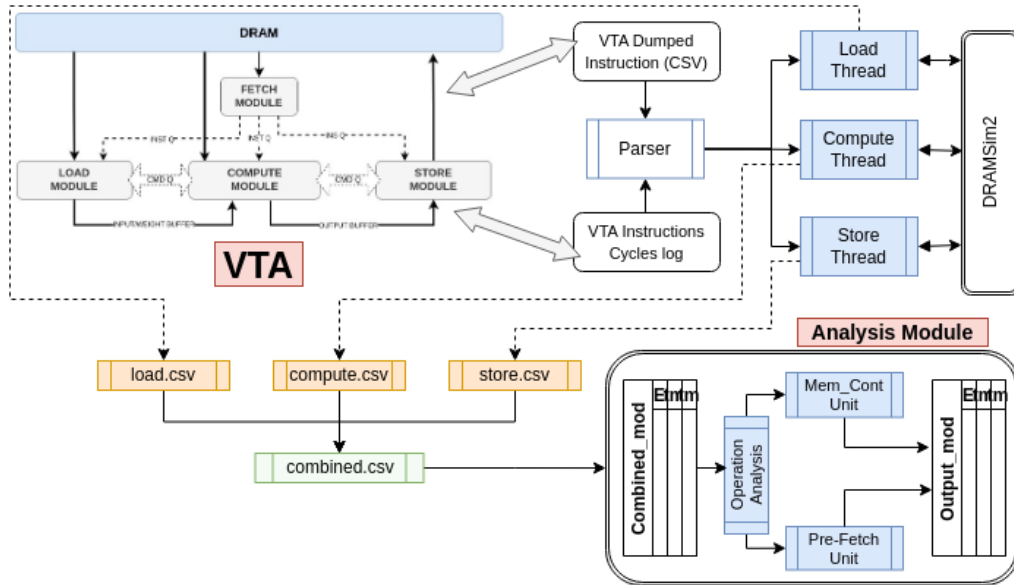


Figure 4.2: Versatile Tensor Accelerator (VTA) Simulator Extension

To capture and analyse the simulator's operations and study its behaviour, the VTA simulator architecture has been extended, as shown in Figure 4.2, by modifying the module threads and introducing a new postprocessing module:

- **Module Threads:** The simulator incorporates module threads adapted to record the timing of memory operations and export the data into module-specific files. Utilising the VTA performance data as input, the extensions in the simulator follow the approach outlined in Algorithm 6 to generate an output database, facilitating further analysis of VTA operations. The initial output of the simulator, enhanced by these extensions, is a database containing logged analytical parameters, such as *log_parameter*, including *inst.layer*, *inst.name*, *rel_start_time*, *rel_stop_time*, *inst.dep*, *inst.start_time*, *inst.finish_time*, *rel_start* (relative start time), and *rel_stop* (relative stop time). The generated data is then exported into designated modulespecific files for post-processing.

Algorithm 6 Extended Simulator Logic

```

1: procedure COMBINE_SORT_MODULES
2:   Open files: l_mod.csv, c_mod.csv, s_mod.csv
3:   Record start time: base_time  $\leftarrow$  inst.start_time
4:   for each inst in inst_queue do
5:     if inst.type  $\neq$  0x00000000 then
6:       rel_start_time  $\leftarrow$  inst.start_time - base_time
7:       rel_stop_time  $\leftarrow$  inst.finish_time - base_time
8:       Append log_parameter to corresponding files l_mod.csv, c_mod.csv, s_mod.csv
9:     end if
10:  end for
11:  Write data to separate .csv files
12:  Close files: l_mod.csv, c_mod.csv, s_mod.csv
13:  Open file: combined_mod.csv
14:  for layer from 0 to 18 do
15:    for each iteration n per layer do
16:      Re-open l_mod.csv, c_mod.csv, s_mod.csv
17:      while there are lines to read from l_mod.csv, c_mod.csv, s_mod.csv do
18:        Read a line from the current input file
19:        Parse line data into log_parameter
20:        if current_layer = layer and current_iteration = n then
21:          Write parsed data to combined_mod.csv
22:        end if
23:      end while
24:    Close l_mod.csv, c_mod.csv, s_mod.csv
25:  end for
26: end for
27:  Close file: combined_mod.csv
28: end procedure

```

- **Post-processing Module:** The postprocessing module aggregates and analyses the separate module data generated by the simulator's thread component. It reads these files (*load_mod.csv*, *compute_mod.csv*, and *store_mod.csv*), processes and organises them in order of occurrence, and combines them into a temporal database. The module then refines the merged data for analysis based on execution order and timing, resulting in a second-stage output database represented as '*combined_mod*'. Additionally, the extension of the simulator allows for the inclusion of an analysis module that examines this database for memory correlation patterns, particularly those involving idle phases of the memory bus. It further evaluates the *log_parameter* to identify opportunities for memory access optimisations. Finally, the simulator's

scheduling module gathers time-based memory access data based on the various optimisations applied to generate optimal memory access schedules.

4.4 Optimising Memory Access and Throughput with the Extended VTA Simulator

This section explores various memory access optimisation techniques integrated into the extended VTA simulator. Building on the enhancements introduced in Section 4.3.1, the focus shifts to addressing identified memory inefficiencies and improving performance through targeted strategies. These strategies, implemented within the analysis module, systematically reduce memory delays, manage idle times, and eliminate contention, resulting in increased overall memory throughput. The following subsections detail each optimisation, demonstrating the practical improvements achieved through simulation-based memory access analysis.

4.4.1 Simulation-based Analysis of VTA Memory Operations

The VTA Simulator has been enhanced with an analysis module integrated within the post-processing module. This analysis module examines the *combined_mod* using a software-based execution timeline analysis technique. This approach allows structured data to be handled efficiently, enabling a comparative analysis of memory delays and latencies to identify optimisation opportunities. After collating the operational data from the VTA simulator, the analysis module comprehensively evaluates the statistical data within the *combined_mod*, aiming to pinpoint potential areas for optimisation. The investigation’s initial focus is assessing memory access dynamics, offering critical insights into how memory is accessed during neural network execution on the VTA.

The output file (*combined_mod*) generated by the VTA software simulator contains execution metrics, including start and finish times, execution duration, layers, operation/instruction names, data types (read and write), memory addresses, and relative time. This output is subjected to further analysis by the VTA Simulator to refine and enhance performance. Initially, the analysis module augments the *combined_mod* by adding additional data, such as absolute timing, execution duration, memory operation delays and latencies, concurrent memory operations, identification flags for memory read-write pair dependencies, utilised memory regions (idle phases timing), and interrelated memory operation markers. This enrichment of data aids in determining appropriate mechanisms that optimise memory access while ensuring the integrity of VTA operations and the fulfilment of instruction dependencies. This statistical framework of the analysis module provides a comprehensive overview of memory access patterns and execution behaviour across different layers. A custom software analysis tool, integrated with the analysis module, is utilised to scrutinise the collected trace data, focusing specifically on memory accesses and access patterns. This custom software tool is designed for simulation-based detailed examination of the statistical data to provide comprehensive insights into memory utilisation throughout a typical neural network’s execution.

The analysis highlights memory access patterns, offering insights into VTA operations and identifying areas that require optimisation. Examining the observed memory access patterns alongside the augmented *combined_mod* data provides critical insights into the model’s execution behaviour, corroborating the findings and observations made during hardware-based tracing, as discussed in Section 3.4.3. These insights guide the selection of effective memory access optimisation strategies.

The issues identified during this analysis emphasise the need to manage concurrent memory operations, address memory waiting periods, and reduce memory idle times, reaffirming the findings of the hardware-based approach in Section 3.4.3. These subsequent actions aim to optimise memory access by reordering execution sequences based on specified access times that align with the strategic simulation activity. A detailed examination of the results of the analysis reveals opportunities to develop a comprehensive strategy to optimise memory access and management during computational phases.

The optimisation process, designed to address the identified issues, mirrors the proposed strategy for optimising memory accesses during computational phases, as discussed in Section 3.4.3. This strategy includes three essential procedures: eliminating memory contention, implementing data prefetching from the main memory during idle phases, and caching write operations with repetitive memory access to the same memory location.

These three optimised solutions have been further adapted to the extensions made to the VTA simulator. These solutions explore advanced strategies to mitigate concurrent memory accesses that lead to contention, manage idle times, address dependency-bottlenecks as waiting periods, and optimise memory accesses through caching. These modifications are integral to simulating and evaluating the effects and outcomes of the strategies aimed at addressing the identified issues. Finally, optimised schedules that incorporate all optimisation solutions are generated, suitable for a time-triggered hardware implementation, ultimately improving memory throughput. The optimised schedule, which is the final output from the extended VTA simulator, is utilised to test and evaluate the memory access enhancements.

4.4.2 Memory Contention Elimination

Effective memory management is critical to enhancing overall system performance. Further scrutiny is applied to memory operations through the analysis module of the simulator, categorising non-zero addresses as memory operations. This preliminary sorting for ResNet-18 on VTA reveals that out of the 4575 operations, 55% are memory operations, with 70% of these operations dependent on non-memory operations. The integrated analytical software tool identifies memory operations that access the memory bus concurrently by comparing their start and finish timings, similar to the observation in ResNet-18 layer one plot of Figure 3.9. Memory contention and potential conflicts during memory access are observed through overlapping periods among memory operations. Findings show that 30% of memory operations overlap, leading to 51% of the total memory operation time being spent queuing on the bus for other operations to complete, confirming the presence of memory contention. The "Memory Contention Elimination" procedure introduced in this section directly addresses the issues stemming from the overlap of memory operations, identified where the finish time of one operation overlaps with the start

time of the next.

The initially proposed overlap detection pseudocode in Algorithm 1 is extended with techniques that incorporate predecessor operations and dependencies to achieve Algorithm 7 that fits within the modified VTA Simulator. This algorithm orchestrates memory operations to ensure a coordinated execution flow where non-memory operations continue without interruption, while memory operation timings are adjusted to avoid concurrent access to the memory bus. The analysis module effectively reduces memory contention by applying the technique outlined in Algorithm 7, thereby addressing the challenges arising from the concurrent execution of memory operations. This contention-free state is achieved by identifying cases within the *combined_mod* where the finish time (*MemOpFinishTime*) of the current operation overlaps with the start time (*MemOpStartTime*) of the subsequent operation, signalling potential contention. The algorithm then adjusts the timing to ensure that these memory operations do not access the bus simultaneously, maintaining a sequential execution flow. This alignment is accomplished by ensuring that each operation's *MemOpStartTime* is synchronised with the *MemOpFinishTime* of the previous memory operation, considering its duration (*ExeDuration*) and dependencies. This procedure ensures that the subsequent memory operation commences after the preceding one concludes and then updates the *combined_mod* to reflect the revised start times for all operations.

Algorithm 7 Eliminating Memory Contentions

```

1: procedure MEMORYCONTENTIONELIMINATE
2:   if current MemOpStartTime  $\geq$  previous MemOpFinishTime then
3:     Run current Memory Operation
4:   else
5:     if MemOp has no predecessor and no dependency then
6:       CurrentStartTime = previous MemOpFinishTime
7:       Run next Memory Operation
8:       NextStartTime = CurrentStartTime + ExeDuration
9:     end if
10:    if MemOp has predecessor and / or dependency then
11:      Wait for predecessor or dependent Memory Operation
12:      LastFinishTime = FinishTime of predecessor || dependent Ops
13:      Run next Memory Operation
14:    end if
15:  end if
16: end procedure

```

The Memory Contention Elimination procedure effectively resolves the challenge of overlapping memory operations, streamlining execution to enhance system performance. Potential contentions are identified and resolved through a systematic alignment of the finish time of the current operation with the start time of the next. The algorithm prioritises memory execution ordering to ensure that each operation executes sequentially after the previous one has finished. This sequential execution strategy resolves memory contentions identified during analysis, significantly improving memory system performance. The refined approach reduces pressure on the memory bus, mitigates variable latency, optimises memory bandwidth usage, stabilises system behaviour, enhances system efficiency, and promotes efficient memory utilisation. By managing memory operations based on the established execution sequence from the traces and initiating the next operation only after the comple-

tion of the current one, the overall system performance is significantly improved, eliminating contention.

Further detailed analysis of VTA memory access patterns using the integrated analytical component within the VTA Simulator software reveals considerable potential for further optimisation. The identified opportunities for enhancement mirror the memory idle phase findings in the hardware-based approach discussed in Section 4.2.1, and leverage the utilisation of idle intervals between successive memory operations to perform data prefetching. This strategy aims to optimise idle memory access phases that occur due to waiting periods associated with unfulfilled dependencies of memory operations. The proposed improvements involve the introduction of a prefetch mechanism that exploits the idle periods for operations awaiting dependency resolution.

4.4.3 Optimising Memory Utilisation through Idle Time Management Using a Data Prefetching Mechanism

An in-depth analysis conducted by the simulator’s analysis module on the VTA collated data reveals that approximately 29% of the total memory execution time consists of idle periods. Further examination uncovers underutilised segments within the memory execution timeline, where delays occur due to unresolved nonmemory dependencies. To address this issue, a prefetching technique is introduced. This technique allows data for the subsequent memory operation to be prefetched when specific criteria are satisfied, thereby reducing waiting times.

The simulator’s analysis module identifies periods of underutilised memory operations, particularly where tasks are delayed due to unfulfilled non-memory dependencies. A pseudocode is employed within the simulator to detect subsequent operations without dependency and can commence without relying on previous operations. Upon identification, the timings of these operations are examined, and decisions are made regarding whether to adjust or maintain their execution order. The software assesses the finish time of prior memory operations relative to the current dependent operation (both memory and non-memory dependencies). The execution order is then modified by adjusting the start times of subsequent operations with fulfilled conditions that fit into the potential delay time window.

The prefetching mechanism for upcoming memory operations is employed when specific conditions are met. These conditions include:

- **Completion of Predecessor:** Prefetching is initiated after the predecessor of the subsequent memory operation has been successfully completed.
- **Absence of Unfulfilled Dependencies:** Prefetching is performed when the subsequent memory operation has no outstanding dependencies, and its duration fits within the potential waiting time window, ensuring smooth execution.
- **No Direct Relation with Pending Operations:** The next memory operation should not have a direct relation with any pending memory or nonmemory operations, which is crucial for maintaining the integrity of the prefetching process.

When these conditions are fulfilled, the data for the subsequent memory operation is prefetched using the pseudocode in Algorithm 8, effectively utilising the

potential waiting window. In such scenarios, prefetching the data for the next memory operation becomes a strategic approach. Once the pending memory operation meets its requirements, the subsequent operations can resume seamlessly as soon as the memory bus is free, potentially further reducing memory execution time.

Algorithm 8 Prefetching Data During Memory Idle Periods

```

1: procedure PREFETCHINGDATA
2:   if Previous Memory Operation ends and Next Memory Operation has dependencies then
3:     while not all dependencies are met do
4:       Wait for dependency completion
5:       if memory bus is free then
6:          $NextStartTime \leftarrow FinishTime$  of the dependency
7:         Start the next Memory Operation
8:       end if
9:     end while
10:  else
11:    if a memory idle phase is identified and memory bus is free then
12:      if Memory Operation has an unfulfilled prefetch condition then
13:        Halt the current Memory Operation
14:        while Current Memory Operation dependencies are not met do
15:           $NextStartTime \leftarrow LastFinishTime$ 
16:          Prefetch and cache the subsequent Memory Operation
17:           $NextStartTime \leftarrow LastFinishTime + \text{current duration}$ 
18:          Wait until the dependency of halted operation is met
19:           $NextStartTime \leftarrow FinishTime$  of the dependency
20:        end while
21:        Wait until the dependent task finishes execution
22:         $NextStartTime \leftarrow FinishTime$  of the dependent operation.
23:        Resume the halted Memory Operation
24:      end if
25:    end if
26:  end if
27: end procedure

```

The "Prefetching Data" procedure optimises memory usage through predictive data prefetching. The algorithm identifies unsatisfied dependent operations and halts them to initiate data prefetching during periods of memory bus availability. It strategically leverages the completion times of previous operations to promptly initiate the subsequent operation, thereby minimising idle times and significantly reducing waiting time within the memory system. This prefetching procedure focuses on refining memory access times by proactively utilising free memory intervals to prefetch subsequent memory operations, thus enhancing overall memory access efficiency. This strategy not only reduces waiting times across operations but also fosters an environment of improved memory access efficiency.

The design of this procedure, detailed in the larger pseudocode of Algorithm 11 in the Appendix, aligns with the overarching goal of optimising memory utilisation through strategic idle time management and proactive data prefetching. Additionally, frequent memory access to the same memory region, particularly in systems with limited memory resources, results in redundancy. The following section explores a simulation-based approach that exploits idle times between consecutive operations and integrates a caching mechanism to optimise write-read memory operations, thereby reducing reliance on off-chip DRAM.

4.4.4 Optimising Memory Access with Intermediate Caching of Repetitive Operations

Another optimisation strategy introduced to enhance memory management is intermediate caching, aimed at significantly reducing the frequency of memory accesses to improve the overall memory access efficiency and responsiveness of the VTA. This approach focuses on optimising memory access operations through a caching strategy specifically designed for repetitive memory access patterns. By utilising a custom analysis module integrated into the VTA Simulator, this strategy evaluates memory access data, including addresses, execution start and finish times, and operation types. The goal is to enhance system performance, minimise memory latency, and improve throughput by caching frequently accessed data. The proposed measures are designed to boost the performance of the entire computational framework, ensuring more efficient handling of repetitive memory operations.

4.4.4.1 Caching Strategy for Enhancing Memory Access Efficiency

In previous sections, the memory access information—such as memory access addresses, operation types (read or write), data size, start and finish times, burst transfer rates (including stride, `x_size`, and `y_size`), and frequency of access—was computed and analysed using the analysis module introduced in the VTA Simulator. The VTA Simulator’s analysis module then focuses specifically on comprehensively examining the characteristics of memory access addresses, the frequency of memory access to specific addresses within a given timeframe, start and finish timing information for each operation to evaluate execution durations, the operation type (read or write), the volume of data involved in each operation, and burst transfer characteristics such as stride, `x_size`, and `y_size`, which are relevant for burst transfers.

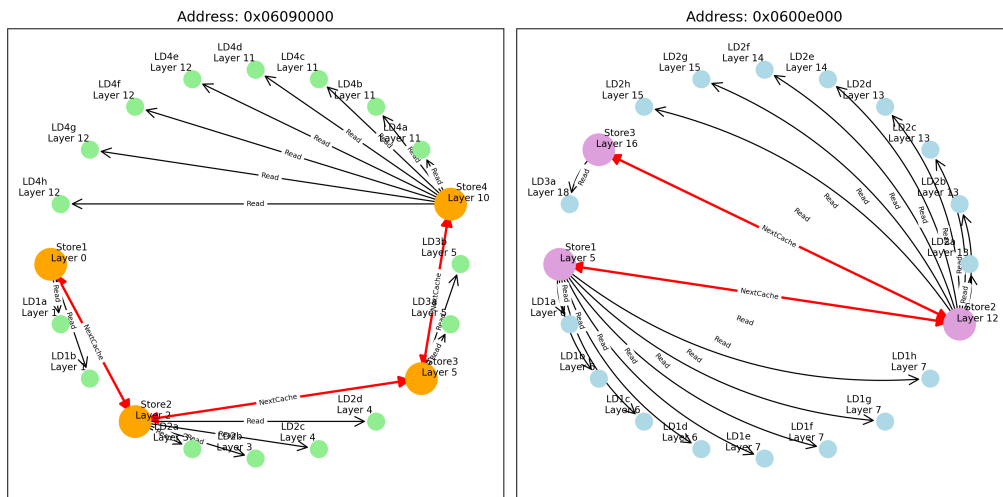


Figure 4.3: A Single Memory Address *Cache Pairs* Graph showing inter-layer Dependency

This analysis identifies patterns where write operations are followed by subsequent read operations to the same memory address location, as depicted in Figure 4.3. These are categorised as "*write-read pairs*" and are also referred to as "*cache*

pairs” in this section. A *cache pair* typically consists of a write operation (denoted as ”Store”) and subsequent read operations (denoted as ”Load”), reflecting data access patterns where memory is written once and read multiple times.

Each *cache pair* can have multiple associated load operations, such as Store 1, LD1a, LD1b, and so on, where LD represents a load (read) operation. The suffixes a, b, c, etc., indicate distinct occurrences of read operations following the same store operation. These suffixes represent different instances where data stored in memory is accessed multiple times during execution. The relationship between store and load operations within a *cache pair* is crucial for understanding memory access patterns, as it highlights both the write-read dependency and the frequency of accesses.

As shown in Figure 4.3, the presence of multiple *cache pairs* for a particular address illustrates how data propagates through various execution layers. These pairs provide insight into the flow of stored values across different operations and time steps, revealing critical memory access dependencies.

These *write-read pairs* are identified by:

- **Write and Read Addresses Analysis:** Address dependencies between VTA memory operations, such as load operations and data written by store operations, are identified.
- **Layer-by-Layer Analysis:** Matching *write-read pairs* are established based on execution sequences, from *layer_i* to *layer_n*, with unique codes assigned to each identical address pair.

Upon identifying the *write-read pairs*, a caching strategy is introduced to optimise data retrieval by caching the write data for each *write-read pair*. Memory bandwidth is then allocated to store data from identified write operations in each cache pair, reducing DRAM access for frequently read operations and freeing the bus for other operations. To manage the caching process efficiently, each *write-read pair* receives a unique cache ID. The cache reference is updated by tracking the address and data size based on the assigned cache IDs for each write operation of a *write-read pair*. The cache reference handles every cache update such that new write operations to the same address invalidate previous cache data and update the cache with new data, continuing the cycle. This caching process eliminates the need for main memory access for subsequent read pairs since the data is already in the cache.

Caching data from write operations alleviates the burden on memory and ensures quick retrieval during subsequent read requests. This caching mechanism utilises the pseudocode in Algorithm 9 to efficiently identify, cache, and manage memory operations. The algorithm, *Intermediate Caching Optimisation for Write-Read Pairs*, implements an optimisation strategy designed to improve the efficiency of repetitive memory access operations by strategically caching the data.

The pseudocode in Algorithm 9 is triggered when the frequency of read operations accessing the same write address reaches or exceeds a benchmark of $n\%$ of the total accesses before the next write operation. This $n\%$ benchmark is determined by considering various factors influencing memory access patterns and system performance.

Algorithm 9 Intermediate Caching Optimisation for Write-Read Pairs

```

1: procedure OPTIMISECACHE(Memory Access Trace)
2:   Initialise CacheID  $\leftarrow 0$ 
3:   Initialise CacheTable  $\leftarrow \{\}$ 
4:   for each operation in Memory Access Trace do
5:     if operation.OperationType == "Write" then
6:       Record WriteAddress, DataSize, WriteTime
7:       CacheTable[WriteAddress]  $\leftarrow \{\text{WriteData}, \text{WriteTime}, \text{CacheID}\}$ 
8:       Increment CacheID
9:     else if operation.OperationType == "Read" then
10:      if WriteAddress  $\in$  CacheTable then
11:        if ReadFrequency  $\geq n\%$  before next Write then  $\triangleright$  where  $n=25$  for ResNet-18
12:          Retrieve CacheID from CacheTable
13:          Redirect Read operation to CacheID
14:          Set EntryBit  $\leftarrow 1$ 
15:        end if
16:        if new Write operation at WriteAddress then
17:          Invalidate previous cache data (CacheTable[WriteAddress]  $\leftarrow \text{None}$ )
18:          Update CacheTable with new Write data and CacheID
19:        end if
20:      end if
21:    end if
22:  end for
23:  for each operation in Memory Access Trace do
24:    if operation is cacheable then
25:      Set EntryBit  $\leftarrow 1$ 
26:    else
27:      Set EntryBit  $\leftarrow 0$ 
28:    end if
29:  end for
30:  Append EntryBit to the updated schedule
31:  Simulate the updated schedule in the VTA Simulator
32:  Compute time savings and performance improvements
33: end procedure

```

The formula in Equation 4.12 is used to determine the optimal caching threshold. This threshold optimises the trade-off between cache utilisation and main memory bus bandwidth efficiency. The optimal frequency threshold Th_{opt} for determining the *write-read pair* cache can be expressed as:

$$Th_{\text{opt}} = \alpha \cdot \frac{f_{\text{read}}(a)}{f_{\text{read}}(a) + 1} + \beta \cdot \frac{D_{\text{size}}(a)}{C_{\text{size}}(a)} + \gamma \cdot \frac{B_{\text{util}}(a)}{1 - B_{\text{util}}(a)} + \delta \cdot \left(\frac{M_{\text{latency}}(a) - R_{\text{latency}}(a)}{W_{\text{latency}}(a)} \right) \quad (4.12)$$

Where:

- Th_{opt} is the optimal threshold percentage for caching.
- $f_{\text{read}}(a)$ is the frequency of read operations to address a before the next write operation.
- $D_{\text{size}}(a)$ is the data size of the memory block at address a .
- $B_{\text{util}}(a)$ is the bus utilisation for address a operations.
- $C_{\text{size}}(a)$ is the cache size allocated for address a .
- $R_{\text{latency}}(a)$ is the latency to access data in the cache.
- $W_{\text{latency}}(a)$ is the latency to access data from the main memory.
- $M_{\text{latency}}(a)$ is the latency of the last memory access to address a .

- $\alpha, \beta, \gamma, \delta$ are weighting factors that influence the cache threshold.

The formula in Equation 4.12 contains four sub-equation terms which include:

- **First Term:** $\frac{f_{\text{read}}(a)}{f_{\text{read}}(a)+1}$ represents the normalised frequency of read operations to a given address. The higher this value, the more often a read operation accesses the same address, suggesting a higher likelihood of caching being beneficial.
- **Second Term:** $\frac{D_{\text{size}}(a)}{C_{\text{size}}(a)}$ reflects the proportion of the data size relative to the cache size. If this ratio is large, it indicates that the cache is more likely to be filled, which should affect the decision to cache based on whether the data can be efficiently stored, considering the limited resources of the target FPGA.
- **Third Term:** $\frac{B_{\text{util}}(a)}{1-B_{\text{util}}(a)}$ accounts for the bus utilisation. Higher utilisation implies that the memory bus is heavily loaded, and caching could alleviate this by reducing bus traffic.
- **Fourth Term:** $\frac{M_{\text{latency}}(a)-R_{\text{latency}}(a)}{W_{\text{latency}}(a)}$ compares the difference between the latency of accessing data from the main memory versus the cache relative to the write latency. A more considerable difference indicates that caching can significantly reduce latency, justifying a lower threshold for caching.
- **Weighting Factors:** α, β, γ , and δ are adjustable parameters that allow tuning of the equation based on system-specific characteristics or priorities, such as prioritising latency reduction or managing cache size constraints.

This formula in Equation 4.12 provides a more nuanced approach to caching decisions, optimising performance based on the actual memory behaviour of the specific application. By calculating Th_{opt} for each address a using existing VTA computed trace data from the simulator, a logical threshold of 25% is determined for ResNet-18, reflecting the specific conditions of the memory access patterns in the system under test. This caching strategy allows subsequent read operations within this *write-read pair* threshold to access data directly from the cache, freeing the memory bus for other operations, speeding up execution, and improving overall memory throughput.

The entire process for the intermediate caching optimisation strategy, based on the pseudocode in Algorithm 9 for repetitive memory access operations, involves the following steps:

1. **Initialisation:** The process begins by initialising a cache identifier (**CacheID**) and an empty **CacheTable** as a reference to store information about cached data, including address, data size, and operation timing.
2. **Process Write Operations:**
 - For each write operation, typically the Store, the algorithm checks if a corresponding read operation exists in the memory access trace, targeting the same memory address, forming a *write-read pair*.

- If the frequency of read operations to the *cache pair* address reaches at least Th_{opt} (25% for ResNet-18) of the total accesses before the next write operation, the algorithm caches the data of the write operation for the *cache pair*.
- The cache is sized based on the address, data size (e.g., 128-bit), and burst transfer rates (including stride, x_{size} , and y_{size}). The cache entry is updated in the **CacheTable** with a starting base address (e.g., 0xA0000000), the assigned **CacheID**, and relevant memory details. The cache depth, C_{depth} , for each *write-read pair* can be calculated as:

$$C_{\text{depth}} = \frac{D_{\text{size}} \times X_{\text{size}} \times Y_{\text{size}}}{S_{\text{stride}} \times A_{\text{width}}} \quad (4.13)$$

The equation C_{depth} calculates the required depth of the cache based on the data size, address width, and burst transfer parameters (stride, x_{size} , and y_{size}).

- The next cache is determined from the previous *write-read pair*. To properly align the cache memory and avoid overlapping operations, the next cache base address $C_{\text{base}}^{(n+1)}$ can then be determined using the current cache base address $C_{\text{base}}^{(n)}$ and the cache depth C_{depth} as follows:

$$C_{\text{base}}^{(n+1)} = C_{\text{base}}^{(n)} + C_{\text{depth}} \quad (4.14)$$

The equation $C_{\text{base}}(n+1)$ is used to determine the base address for the next *write-read pair*'s cache, ensuring proper alignment in memory.

Where:

- D_{size} : Data size of the memory block (in bits), e.g., 128-bit.
- A_{width} : Address width (in bits), e.g., 32-bit.
- S_{stride} : Stride value for burst transfer.
- X_{size} : Number of data elements along the x-dimension.
- Y_{size} : Number of data elements along the y-dimension.
- C_{depth} : Cache depth required for each *write-read pair*.
- $C_{\text{base}}^{(n)}$: Base address for the n^{th} *write-read pair* cache.

3. Process Read Operations:

- For read operations, the algorithm verifies whether the corresponding write address is present in the **CacheTable**.
- If the cached data is available for a read operation that satisfies the caching threshold (Th_{opt}), which is set at 25% in this case for ResNet-18, the read operation is redirected to the cached data, thereby reducing the load on the memory bus.
- If a subsequent write operation targets the same address, the algorithm invalidates the previous cache entry and updates it with the new write data.

4. Cache Management:

- The algorithm assigns each *write-read pair* that meets the caching criteria a **CacheID** and sets an **EntryBit** to 1, indicating that it is cacheable. Non-cacheable operations are marked with an **EntryBit** of 0.
- All operations are analysed to identify those accessing the same address. Operations identified as frequent that meet the threshold, Th_{opt} , are redirected to retrieve data from the cache, thereby minimising repeated memory accesses and enhancing efficiency.

5. Performance Simulation:

- After processing the memory access trace, the updated schedule, including the **EntryBit** values, is simulated in the VTA Simulator. This final simulation, which utilises the optimised schedule, quantifies the time savings and assesses the performance improvements achieved through the caching optimisation technique.

4.4.4.2 Caching Performance Improvement and Time Efficiency Computation

The proposed optimisation strategy systematically optimises intermediate caching for *write-read pairs*, an effective approach in high-performance computing environments, such as neural network processing. The algorithm reduces memory access latency and enhances data retrieval speeds by managing cache entries and evicting outdated data when necessary. This cache eviction is particularly advantageous in neural network memory access scenarios, where certain memory addresses are frequently accessed, allowing for more efficient use of the memory bus and faster execution of subsequent operations.

The empirical assessment provided by using the VTA Simulator to simulate this caching technique ensures that the theoretical improvements outlined in this chapter are effectively translated into practical performance gains. The effectiveness of a caching strategy can be quantified using several key performance metrics. These metrics help quantify the improvements achieved through caching in terms of time savings, cache efficiency, and bus utilisation.

One of the primary metrics is the time saved by caching (T_{save}), which is calculated by measuring the total reduction in read time from the main memory due to caching:

$$T_{save} = \sum_{i=1}^N (T_{read_mem}[i] - T_{read_cache}[i]) \quad (4.15)$$

Here:

- $T_{read_mem}[i]$ represents the time required for reading from the main memory.
- $T_{read_cache}[i]$ denotes the time required for reading from the cache.
- N is the total number of cacheable read operations.

By calculating T_{save} , we can assess how much time is saved by using the cache compared to reading directly from the main memory.

Another important metric is cache efficiency (E_{cache}), which quantifies how effectively the cache is utilised:

$$E_{\text{cache}} = \frac{\text{Number of cache hits}}{\text{Total number of read operations}} \quad (4.16)$$

Cache efficiency provides insight into the proportion of read operations that are successfully served by the cache. Higher efficiency indicates a more effective caching strategy.

Finally, the improvement in bus utilisation (B_{improve}), due to caching can be expressed as:

$$B_{\text{improve}} = \frac{T_{\text{save}}}{T_{\text{total.execution}}} \quad (4.17)$$

Where $T_{\text{total.execution}}$ represents the total execution time without caching. This metric measures the relative improvement in system bus utilisation achieved by implementing the caching strategy.

The metrics T_{save} , E_{cache} , and B_{improve} collectively assess the impact of caching on memory performance. T_{save} captures the time reduction, E_{cache} evaluates the effectiveness of cache usage, and B_{improve} measures the enhancement in bus utilisation. However, to further evaluate the performance improvements and compute time efficiency gains from the caching optimisation, three additional computations are made, including S_{time} , $T_{\text{total}}(\text{cache})$, and $P_{\text{improvement}}$.

S_{time} represents the time savings achieved by caching, calculated as the difference between the time required for memory reads from the main memory and the cache. The time savings S_{time} from using the cache can be expressed as:

$$S_{\text{time}} = N_{\text{writes}} \times (T_{\text{read.mem}} - T_{\text{read.cache}}) \quad (4.18)$$

$T_{\text{total}}^{(\text{cache})}$ gives the total execution time with caching, and $P_{\text{improvement}}$ provides the percentage of performance improvement, which quantifies the effectiveness of the caching strategy. The total execution time with caching $T_{\text{total}}^{(\text{cache})}$ is given by:

$$T_{\text{total}}^{(\text{cache})} = T_{\text{total}}^{(\text{no.cache})} - S_{\text{time}} \quad (4.19)$$

Finally, the performance improvement percentage $P_{\text{improvement}}$ can be computed as:

$$P_{\text{improvement}} = \frac{S_{\text{time}}}{T_{\text{total}}^{(\text{no.cache})}} \times 100\% \quad (4.20)$$

This formula quantifies the improvement in execution time attributable to the caching mechanism, reflecting the overall performance gains achieved by the optimisation strategy. Where:

- $T_{\text{read.mem}}$: Time taken to perform a read operation from main memory.
- $T_{\text{read.cache}}$: Time taken to perform a read operation from the cache.
- $T_{\text{write.mem}}$: Time taken to perform a write operation to main memory.
- $T_{\text{total}}^{(\text{no.cache})}$: Total execution time without caching.

- $T_{\text{total}}^{(\text{cache})}$: Total execution time with caching.
- S_{time} : Time savings due to caching.
- $P_{\text{improvement}}$: Performance improvement percentage.
- N_{writes} : Number of *write-read pairs*.

To fully capture the benefits of caching across all memory operations, the time saved from both cacheable read operations and write-read pairs are combined to give the total time savings due to caching.

Given that both metrics, T_{save} and S_{time} , ultimately measure time savings from caching, a unified metric could combine both to reflect total caching-related time savings across all memory accesses (both read and write-read pairs), represented by $T_{\text{cache_total}}$:

$$T_{\text{cache_total}} = T_{\text{save}} + S_{\text{time}} \quad (4.21)$$

Where:

- T_{save} reflects the time saved for cacheable read operations.
- S_{time} reflects the time saved due to caching in write-read pairs.

The expanded equation is given as:

$$T_{\text{cache_total}} = \sum_{i=1}^N (T_{\text{read_mem}}[i] - T_{\text{read_cache}}[i]) + N_{\text{writes}} \times (T_{\text{read_mem}} - T_{\text{read_cache}}) \quad (4.22)$$

The expanded equation, accounting for various factors such as cache hit/miss penalties and different weights for read and write operations, is given by:

$$T_{\text{cache_total}} = \sum_{i=1}^N (\alpha_{\text{read}} \cdot (T_{\text{read_mem}}[i] - T_{\text{read_cache}}[i]) + \beta_{\text{penalty}} \cdot T_{\text{miss}}[i]) + N_{\text{writes}} \times (\alpha_{\text{write}} \cdot (T_{\text{write_mem}} - T_{\text{write_cache}}) + \gamma_{\text{latency}} \cdot T_{\text{latency}}) \quad (4.23)$$

Where:

- α_{read} and α_{write} : Weights for read and write operations, which could reflect different memory access priorities or costs.
- β_{penalty} : Penalty factor for cache misses.
- $T_{\text{miss}}[i]$: Time penalty incurred for cache misses for the i -th memory access.
- γ_{latency} : Latency factor representing additional delays in the system, such as bus or memory controller latency.
- T_{latency} : Overall memory access latency.

This metric provides a more comprehensive understanding of the performance gains by representing the total time saved through caching across different types of memory accesses in the system. These computed gains validate the algorithm’s effectiveness in optimising memory access patterns through intelligent caching strategies, thereby enhancing computational efficiency and performance in computing architectures.

Finally, the complete caching pseudocode in Algorithm 12 is integrated into the VTA Simulator, with the updated memory access schedule reflecting the cacheable operations. The simulation adjusts start and stop times based on the reordered execution and dependencies, optimising overall memory execution with reduced inference time. The updated schedule, including **EntryBits** indicating cacheable operations, is executed and tested on the simulator to evaluate performance.

Simulation results are analysed to determine improvements in time savings, cache efficiency, and bus utilisation. The system reduces memory access latency by caching frequently accessed data and improves overall performance. This detailed strategy for optimising memory access operations through intermediate caching of *write-read pairs* effectively reduces memory latency, alleviates bus congestion and enhances throughput. The detailed algorithm and mathematical formulations provide a robust framework for implementing and evaluating the caching strategy on the VTA Simulator for neural network processing, such as ResNet-18.

All the proposed optimisation strategies in this chapter are implemented as integrated procedures into the simulator using a **flag_reference** for each optimisation process, based on the pseudocode referenced in Algorithms 7, 11, and 12. These strategies include memory data capturing and analysis, idle time management, frequently read data caching, and prefetching under specific conditions. By leveraging the VTA Simulator to simulate the behaviour of the optimised processes, three distinct schedules—**load.sched**, **compute.sched**, and **store.sched**—are produced. Each schedule represents the optimised execution for the load, compute, and store modules.



Figure 4.4: Optimised Schedule Frame

The updated data is then converted into optimised 64-bit schedules, as depicted in Figure 4.4, using the adapted scheduling pseudocode provided in Algorithm 2. This updated scheduling process ensures that the reordered execution sequence, which reflects the integrated process approaches, is enforced. The final optimised schedules incorporate a prefetch flag and a cache flag to identify which operations require additional activities. These schedules are then tested by comparing the original traced input execution order and timing of the framework proposed by [150] with the updated and optimised schedule data to evaluate performance.

These optimised procedures collectively provide a base platform for the VTA with Optimised Time-Triggered Memory Access, otherwise named as **OpTTmaVTA**, which outperforms the VTA in terms of predictability, inference timing, and enhanced memory management. It eliminates contention, employs data prefetching to reduce avoidable memory idleness, and optimises memory accesses for *write-read*

pairs through caching. The resulting schedules—`load.sched`, `compute.sched`, and `store.sched`—reflect the performance improvements achieved through these strategies and are valuable for advancing the time-triggered implementation in hardware to translate the simulated design and schedules into a practical implementation of OpTTmaVTA.

Chapter 5

Experimental Setups, Test Evaluations, and Results

In Chapter 3, the design and methodology of the time-triggered memory access VTA (TTmaVTA) were introduced. This architecture enables the scheduled operations of the VTA to be executed in a time-triggered manner, thereby improving system predictability and ensuring timing guarantees in current AI accelerators. Additionally, Chapter 4 detailed the memory optimisation strategies, emphasising efficient memory management through data prefetching by executing subsequent memory operations during anticipated idle periods, caching frequently accessed data, mitigating memory contention, and optimising memory bandwidth to enhance memory throughput. In this chapter, the experimental setup used to evaluate the performance of these strategies by assessing the effectiveness of the proposed algorithms in managing memory access and optimising memory execution sequences is outlined for both TTmaVTA (in hardware experiments) and its optimised version, OPTTmaVTA, (in software simulation). Their performances are assessed based on metrics such as latency, execution time, memory access efficiency, and overall time improvements.

This chapter further evaluates how these optimisations—prefetching during idle memory phases, caching frequently accessed data, and reducing contention—improve system performance. The TTmaVTA, as presented in Chapter 3, is evaluated in a hardware setting, as detailed in Section 5.1, while the OPTTmaVTA discussed in Chapter 4 is evaluated using a software-based simulator in Section 5.3 and Section 5.4. These evaluations provide comprehensive insights into the improvements achieved through the time-triggered scheduling approach and memory access optimisation strategies.

5.1 Hardware-Based Experimental Evaluation (TTmaVTA)

This section presents the performance evaluation of the TTmaVTA in a hardware-based setting, specifically focusing on its performance when executing an optimised 2D Convolutional Neural Network (Conv2D) model. The primary objective is to evaluate memory access patterns, memory access time predictability, and overall memory access time within a time-triggered scheduling system.

5.1.1 Memory Access Tracing and Time-Triggered Scheduling

Memory access tracing and analysis have been prior conducted to capture detailed logs of memory access and read/write operation sequences. These logs were utilised to analyse memory access patterns, identify inefficiencies such as memory access contention, and generate an efficient time-triggered schedule. From this study, the VTA executes its instructions sequentially across various modules, which facilitates a detailed examination of memory-related tasks by observing memory patterns and the behaviour of the VTA. Key insights from memory access tracing include:

- Memory accesses exhibited periodic collisions due to concurrent operations initiated by multiple memory tasks.
- Without an optimised schedule, memory access latency variability arose from factors such as contention during concurrent memory operations attempts and the fulfilment of dependencies. The time-triggered scheduling approach mitigated these.
- Approximately 39.27% of memory idle time is observed, presenting significant opportunities to optimise access patterns to maximise memory throughput while maintaining system predictability.

Analysing memory access patterns provided insights for creating an appropriate time-triggered schedule for memory accesses. Each memory access was assigned a specific operating time slot to prevent concurrent memory accesses, causing contentions. The time-triggered controller (TTmaC) managed dispatching memory-related tasks based on pre-determined time intervals. The scheduling approach described in Section 3.5 ensured that each access occurred predictably, without concurrent execution attempts. The schedule parameters include:

- **Memory Access Time (MAT):** The latency for each memory operation was recorded through multiple test runs. These data were used to construct the MAT table, which is deployed to ensure predictable and deterministic memory access. Each memory access was allocated a 28-bit time slot, with the operation encoded into a 2-bit identifier (read/write).
- **Memory Access Address:** The 32-bit memory address was stored in the schedule entry to reference the memory location for data access.

The time-triggered scheduling mechanism, utilising Algorithm 8, ensured that memory accesses occurred within designated time windows, eliminating contention and reducing latency variations observed in the unscheduled memory accesses of the conventional VTA.

5.1.2 Hardware Setup and Testbench Configuration

The hardware platform comprises a Trenz Electronics TE0802 board featuring the Xilinx Zynq UltraScale+ MPSoC FPGA, equipped with a dual-core processor and 1 GByte of LPDDR4 memory. The TTmaVTA, implemented as a hardware configuration file in the form of a bitstream (.bit), is deployed on the Programmable

Logic (PL) of the TE0802 board, with the pre-computed schedule for memory operations loaded into the FPGA. The testing environment utilises PYNQ-Linux on an Ubuntu 18.04 base system, running kernel version 5.4.0, which includes modifications for software cache coherency and memory isolation.

The Conv2D testbench, employed as a workload, is executed in repeated runs to capture memory access traces and patterns. The Conv2D model is deployed on the FPGA from the host machine via Remote Procedure Call (RPC) along with the TTmaVTA hardware configuration file. The Time-Triggered Memory Access Controller (TTmaC) of the TTmaVTA managed the memory operations. The baseline VTA (without time-triggered scheduling) completes the task in 20.54 ms using the existing default VTA hardware configuration file. The TTmaVTA architecture is subsequently applied to the same workload for comparison to evaluate the improvements in terms of memory contention, memory access efficiency, and overall reduction in memory access time.

5.1.3 Addressing Memory Contention in VTA Memory Operations

Efficient memory utilisation is crucial for optimising system performance. The VTA trace analysis in Section 3.3.2 uses Algorithm 1 to categorise operations based on memory access. Of the total 4,575 traced operations, 55% are identified as memory-bound, with 70% of these operations exhibiting dependencies on non-memory tasks. The analysis further revealed that 42.5% of memory operations concurrently attempt to access the memory bus, resulting in significant contention.

The VTA trace analysis, discussed in Section 3.6, affirmed memory contention as a critical bottleneck in optimising system performance. The analysis identified 1,072 instances where memory operations attempted to access the bus concurrently, leading to queuing delays. Approximately 30% of the total execution time for memory operations is spent in queues, waiting for bus access. This contention at the bus highlights the necessity for more efficient memory access handling.

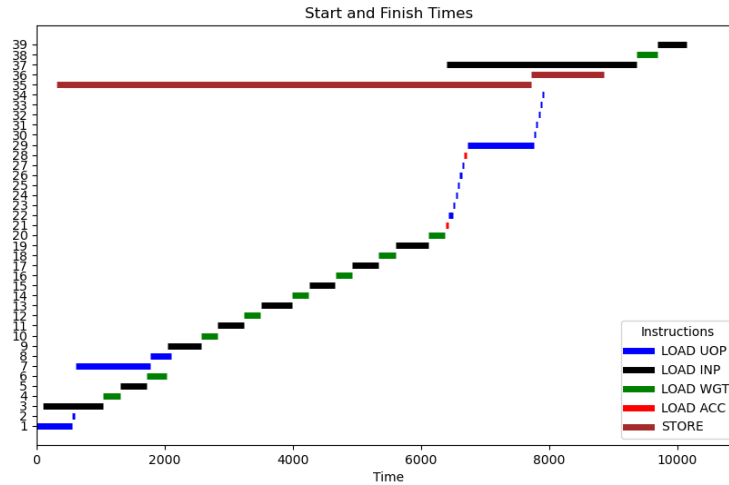


Figure 5.1: Memory Operation Execution for VTA Layer-1: Stage-1

Memory contention in this context arises from overlapping traced periods where

multiple memory operations attempt to access the memory bus simultaneously, leading to resource contention and a reduction in performance efficiency. Figure 5.1 illustrates the memory contention detected in the first layer of VTA operations, where concurrent operations such as a **STORE** operation overlap with **LOAD** operations, resulting in significant memory access inefficiency. The memory operation sequence for the first layer of the VTA, as shown in Figure 5.1, confirms the presence of substantial contention. For example, the second-stage **STORE** operation (marked in brown) attempts to execute concurrently with the **LOAD INP** operation (marked in black). To resolve this issue, memory operation timings are realigned to avoid contention, utilising the methods and algorithms outlined in Chapter 3, ensuring that each operation starts only after the previous one has been completed.

The pseudocode of Algorithm 1 details the technique for eliminating memory contention by carefully orchestrating the access times of memory operations, taking dependencies and predecessor tasks into account. By aligning each memory operation’s start time with the previous operation’s completion, the algorithm addresses the risk of contention and promotes efficient memory utilisation. The execution timing of memory operations is adjusted to prevent overlapping access periods. The proposed approach maintains the order of execution while ensuring that each operation is completed before the next begins. This method effectively resolves the issue of concurrent memory operations, particularly cases where one operation’s completion coincides with another’s start. This solution ensures orderly execution, mitigates contention and improves overall memory system performance. Figure 5.2 shows the memory access trace after contention has been eliminated. All memory operations are now executed sequentially, with no overlaps between **STORE** and **LOAD** operations.

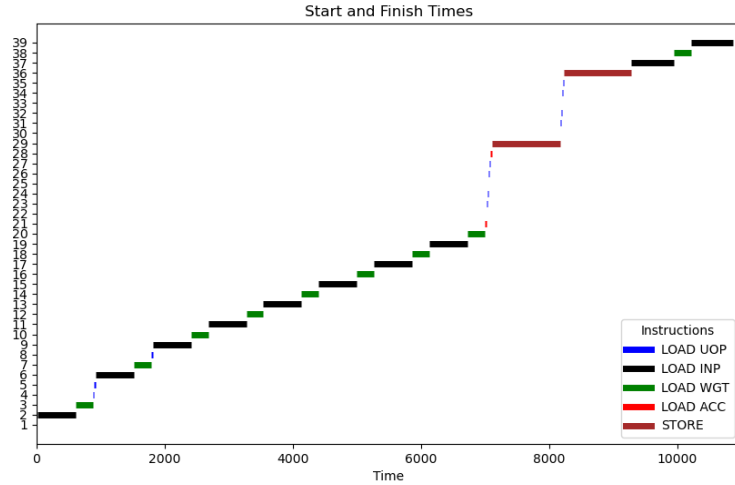


Figure 5.2: Optimised Memory Access Operation without Contention

A performance evaluation comparing the memory operations before and after applying the contention mitigation strategy is presented in Figure 5.1 and Figure 5.2. The plot of Figure 5.1 depicts the baseline VTA execution of memory operations, where overlaps occurred, leading to conflicts—such as the second **STORE** operation overlapping with the **LOAD INP** operation. However, Figure 5.2 shows the sequential execution of all memory operations after contention mitigation, with no overlaps,

effectively addressing resource contention and minimising latency variability. This restructured execution mitigates contention and improves system performance.

The proposed approach successfully mitigates memory contention by realigning memory operations to ensure sequential execution, as demonstrated in Figure 5.2, enabling smooth, conflict-free operation of memory tasks. The performance of TTmaVTA was evaluated against the conventional VTA (also referred to as the baseline VTA) in multiple test cases. The results indicate that TTmaVTA outperforms the baseline VTA by offering predictable memory access patterns, eliminating contention, and enforcing sequential memory access without overlapping execution attempts. This evaluation demonstrates that TTmaVTA significantly mitigates memory contention, leading to improved memory throughput.

5.1.4 TTmaVTA Performance Analysis

The performance of TTmaVTA and the baseline VTA were examined and compared using key metrics such as execution time, memory throughput, and latency. The TTmaVTA framework demonstrated modest improvements in these performance metrics, primarily due to eliminating memory contention and scheduling memory operations to operate in a time-triggered manner.

5.1.4.1 Execution Time

The TTmaVTA framework demonstrated a reduction in execution time compared to the conventional VTA, with a 2.87% improvement. This reduction is attributed to prefetching data ahead of VTA module requests, mitigating memory access collisions, and optimising memory access patterns through the time-triggered scheduling system. However, the performance improvements shown in Table 5.1 are more modest than initially anticipated, potentially due to the additional resource overhead introduced by the TTmaC.

Framework	Average Execution Time (ms)	Improvement (%)
Conventional VTA	20.54	-
TTmaVTA	19.95	2.87%

Table 5.1: Execution Time Comparison between Conventional VTA and TTmaVTA

The results indicate that while the time-triggered scheduling mechanism mitigates memory contention and ensures predictable execution, the additional overhead from the TTmaC imposes minor performance limitations.

5.1.4.2 Memory Throughput

Memory throughput, which measures the efficiency of data transfer between memory and processing cores, shows a notable improvement with the time-triggered scheduling mechanism, as seen in Table 5.2. The memory throughput is calculated as follows:

$$\text{Memory Throughput (MB/s)} = \frac{\text{Total Data Transferred (MB)}}{\text{Execution Time (s)}}$$

For a total data transfer of 40 MB, the throughput values are as follows:

$$\text{Memory Throughput (Conventional VTA)} = \frac{40 \text{ MB}}{20.54 \text{ ms}} = 1.95 \text{ GB/s}$$

$$\text{Memory Throughput (TTmaVTA)} = \frac{40 \text{ MB}}{19.95 \text{ ms}} = 2.00 \text{ GB/s}$$

Framework	Memory Throughput (GB/s)	Improvement (%)
Conventional VTA	1.95	-
TTmaVTA	2.00	2.56%

Table 5.2: Memory Throughput Comparison between Conventional VTA and TTmaVTA

Memory throughput improves due to the reduction in execution time, with the same data transfer amount in both frameworks. The modest execution time reduction resulted in a throughput increase of 2.56%.

5.1.4.3 Latency Analysis

Latency, representing the time taken to service memory requests, also exhibited slight improvement with the TTmaVTA framework. However, the increased complexity introduced by the TTmaC had a marginal impact on these gains.

Framework	Average Latency (ns)	Improvement (%)
Conventional VTA	1050	-
TTmaVTA	1020	2.86%

Table 5.3: Latency Comparison between Conventional VTA and TTmaVTA

The results in Table 5.3 indicate a 2.86% reduction in latency with TTmaVTA, though the additional resource utilisation of the TTmaC constrains this improvement.

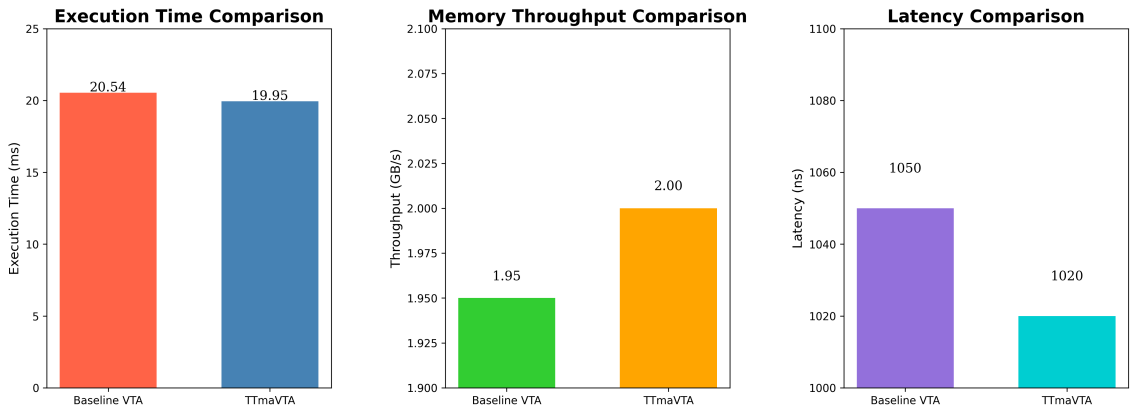


Figure 5.3: Performance Comparison of Memory Access Time, Memory Throughput, and Latency between Conventional VTA and TTmaVTA.

Figure 5.3 visually represents the performance comparison for the improvements observed in execution time, memory throughput, and latency between TTmaVTA and the baseline VTA. The plot compares the key performance metrics of execution time, memory throughput, and latency.

5.1.4.4 Memory Accesses Latency Computation and Comparison between Baseline VTA and TTmaVTA

The maximum and minimum latencies of memory accesses are computed with data from multiple test runs of the TTmaVTA. These calculations are based on the total memory duration between the baseline VTA and TTmaVTA frameworks.

Maximum Latency (MaxL)

The maximum memory latency is given by:

$$\text{MaxL} = \max\{L_1, L_2, \dots, L_N\} \quad (5.1)$$

Where:

$L_1, L_2, \dots, L_N$ represent the set of memory access latencies, and N is the total number of memory accesses.

Minimum Latency (MinL)

The minimum latency corresponds to the cycles spent during active memory operations, excluding idle periods. It is expressed as:

$$\text{MinL} = \min\{L_1, L_2, \dots, L_N\} \quad (5.2)$$

Where:

MinL represents the minimum value from the set of memory latencies $L_1, \dots, L_N$.

The execution of Conv2D across 100 iterations recorded a total memory execution time for the baseline VTA with a maximum memory latency of 217,724 cycles and a minimum memory latency of 212,001 cycles. In contrast, TTmaVTA demonstrates consistent memory latency at 211,469 cycles due to its time-triggered scheduling approach. The maximum and minimum memory access latencies of the baseline VTA given by MaxL_{VTA} and MinL_{VTA} and for the TTmaVTA given by $\text{MaxL}_{\text{TTmaVTA}}$ and $\text{MinL}_{\text{TTmaVTA}}$ are:

$$\text{MaxL}_{\text{VTA}} = 217,724 \text{ cycles}, \quad \text{MinL}_{\text{VTA}} = 212,001 \text{ cycles} \quad (5.3)$$

$$\text{MaxL}_{\text{TTmaVTA}} = \text{MinL}_{\text{TTmaVTA}} = 211,469 \text{ cycles} \quad (5.4)$$

The baseline VTA exhibits fluctuations in memory latency due to factors such as memory contention irregularities, uncertainties in memory accesses, inconsistencies during idle periods, and overall latency variability, while the TTmaVTA consistently operates at lower latency due to its predefined memory scheduling.

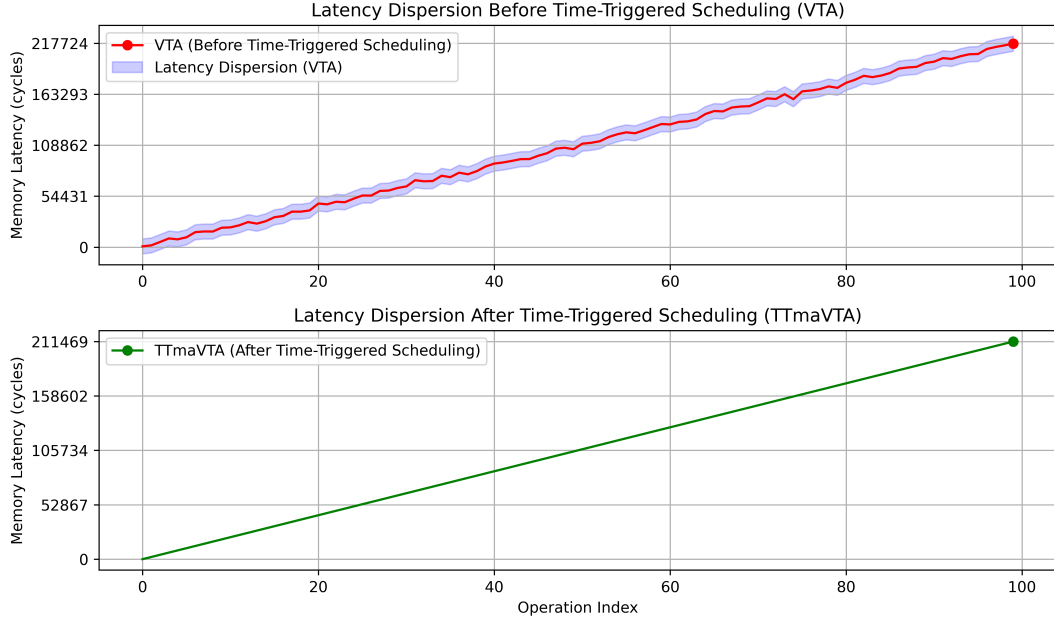


Figure 5.4: Latency dispersion curve comparison between VTA and TTmaVTA.

Figure 5.4 illustrates the latency dispersion curves for the baseline VTA and TTmaVTA. The plot highlights the performance improvements of TTmaVTA, demonstrating reduced overall latency compared to the baseline VTA across different memory operation sizes and transfer rates. Additionally, the plot contrasts the baseline VTA with TTmaVTA, the latter employing predefined memory access schedules to maintain a consistent latency through a pre-computed schedule. This plot shows reduced latency and a more stable performance for the TTmaVTA. This comparison underscores the impact of time variability in the baseline VTA, in contrast to the deterministic nature of time-triggered scheduling in TTmaVTA, highlighting the advantages of using time-triggered architectures for memory accesses.

5.1.4.5 Resource Utilisation Analysis

The TTmaVTA system demonstrated improved resource utilisation, particularly in cache memory and memory controllers.

TTmaVTA effectively alleviates strain on these resources by scheduling memory accesses and mitigating contention. The time-triggered approach reduces memory access conflicts, resulting in smoother data flow, particularly in cache memory and SRAM. Although the shared cache used by the TTmaVTA is optimally sized to minimise fetch delays without causing synthesis or routing issues, introducing the Time-Triggered Memory Access Controller (TTmaC) results in additional resource consumption.

Figure 5.5 compares resource utilisation percentages for both VTA and TTmaVTA architectures, highlighting the trade-offs between improved memory management and increased resource overhead. The increased resource consumption may limit the scalability of the scheduling approach when utilising small-to-medium scale FPGAs for larger models or more complex workloads.

Resource	VTA Utilization	TTmaVTA Utilization	Available	VTA Utilization (%)	TTmaVTA Utilization (%)
LUT	26865.0	38955.0	47232	56.88	82.48
LUTRAM	2195.0	2359.0	28800	7.62	8.19
FF	34340.0	55430.0	94464	36.35	58.68
BRAM	136.5	141.8	150	91.0	94.53
DSP	240.0	240.0	240	100.0	100.0
BUFG	2.0	2.0	196	1.02	1.02
MMCM	1.0	1.0	3	33.33	33.33

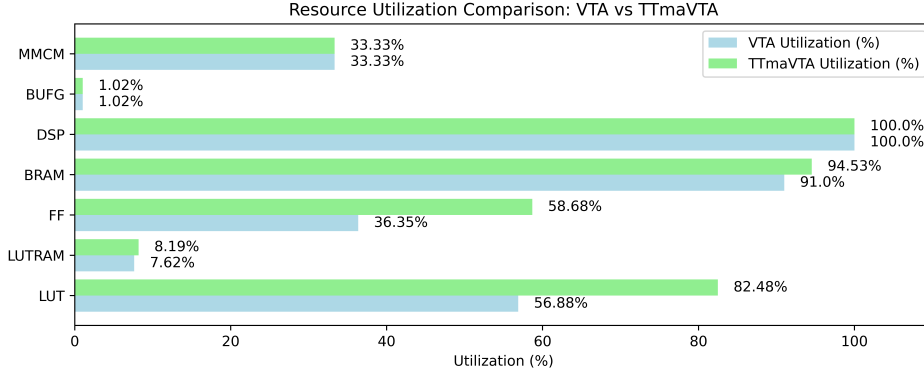


Figure 5.5: Resource utilisation comparison between VTA and TTmaVTA.

5.1.5 Performance Evaluation and Discussion

The TTmaVTA framework exhibited marginal improvements in performance metrics compared to the baseline VTA, particularly in memory latency, memory throughput, and overall execution time. These improvements are primarily attributed to the reduction in memory access contention and the more efficient handling of memory requests through time-triggered scheduling.

The most notable improvement is a 2.87% reduction in execution time, primarily attributed to the reduction in memory access collisions. However, the additional resource overhead introduced by the TTmaC constrained these performance gains. Memory throughput and latency improved, with gains of 2.56% and 2.86%, respectively. While the TTmaC effectively managed all memory requests, the overall benefit was minimal due to the increased system complexity.

The TTmaVTA framework introduces a time-triggered memory access scheduling mechanism that reduces memory contention and ensures the predictability of memory access patterns. The results demonstrate notable improvements in execution time, memory throughput, and latency. However, the resource overhead introduced by the TTmaC limits the overall performance gains.

While the improvements in memory management are evident, the Conv2D workload used in the evaluation did not exhibit significant memory contention, reducing the observable performance gains. Larger models, such as ResNet, may benefit more from this approach. However, scalability concerns arise due to the resource consumption of the TTmaC, necessitating further optimisation for larger or more complex workloads.

Overall, TTmaVTA presents a structured and predictable approach to memory management for the VTA, with the potential for broader application following resource optimisation and refinement of the time-triggered components.

5.2 Software-Based Evaluation of Optimisation Strategies (OPTTmaVTA)

This section presents a simulation-based evaluation of the VTA with Optimised Time-Triggered Memory Access (OPTTmaVTA). The evaluation uses an extended version of the VTA simulator, equipped with capabilities for detailed memory access analysis and post-processing of traced data, as discussed in Section 4.3. The evaluation focuses on the ResNet-18 model, examining its memory access behaviour and applying optimisation techniques such as memory prefetching and caching frequently accessed data to reduce idle memory phases and improve throughput.

Simulation Configuration

The simulation environment replicates baseline VTA operations, with the addition of detailed trace analysis for memory operations, including latencies and execution durations. This tracing allows for a detailed examination of inefficiencies such as memory contention, idle periods, and repetitive access patterns. The baseline simulation identified several inefficiencies in conventional VTA operation:

- **Memory Access Contention:** Multiple operations simultaneously attempting to access the memory bus caused access delays.
- **Frequent and Repetitive Write-Read Operations:** Repeated access to the same memory location by multiple **LOAD** operations to fetch data written by the same **STORE** operation, adding avoidable pressure on the memory bus.
- **Memory Latency and Idle Phases:** Unfulfilled dependencies during execution resulted in idle memory periods, stalling the memory bus and preventing other operations from proceeding.

The VTA simulator used for this evaluation incorporated a prefetching mechanism to take advantage of idle memory phases, executing subsequent operations during these periods of idleness. Additionally, repeated memory operations identified through matching *write – read* cache pairs were cached to eliminate redundant memory accesses.

Performance Optimisation

The proposed optimisations focus on three core areas:

- **Memory Contentions:** A scheduling algorithm resolves memory contention by aligning memory operations sequentially, similar to the hardware-based TTmaVTA approach discussed in Section 5.1, and therefore it will not be further detailed here.
- **Caching of Repetitive Data:** Identified repeated accesses to the same address were cached to reduce redundant memory operations, leading to an approximate 8.75% reduction in memory access latency, thereby improving memory throughput.

- **Memory Prefetching:** The prefetching mechanism identified idle phases and reordered operations, adjusting start times of subsequent tasks that met prefetch conditions. Prefetching led to a reduction in memory access latency by 12.68%.

The evaluation was conducted using an adapted VTA simulator equipped with a simulation-test module. Various flags were used, embedded in precomputed schedules, as discussed in Sections 5.3 and 5.4, to evaluate the effectiveness of the optimisation techniques. These flags utilised prefetch and caching bits assigned to operations falling into specific categories, allowing for scheduled memory execution without disrupting VTA transaction flow.

5.3 Evaluating Data Prefetching Optimisation Approach

The optimisation of memory idle phases through data prefetching is a key technique for enhancing the overall memory efficiency of the VTA. The approach, outlined in Section 4.2, aimed to improve memory utilisation and streamline execution flow by reducing idle periods during memory tasks. A prefetching strategy was introduced to reorder memory operations, especially rescheduling those with unfulfilled dependencies, to take full advantage of identified idle periods and enhance overall memory execution.

5.3.1 Memory Idle Phase Analysis and Prefetching Strategy

The identified memory idle phases in the VTA were primarily characterised by periods during which memory operations waited to fulfil non-memory dependencies. These phases were optimised by introducing a prefetching algorithm to pre-load data required for subsequent memory operations that met the necessary conditions. The process involved extending the simulator with an analysis module capable of identifying idle periods, defined as gaps between consecutive memory tasks. Figure 5.6 illustrates memory latency and reveals significant idle periods due to unfulfilled dependencies. The plot, extracted from the first layer of memory access in the VTA, highlights idle periods caused by unfulfilled dependencies.

5.3.1.1 Identification and Characterisation of Idle Phases

The initial step in optimising memory access involves identifying idle phases and their underlying causes. A significant portion of these idle periods was attributed to delays in dependency fulfilment, particularly in `LOAD INP` operations, which often awaited the completion of non-memory dependencies, such as `COMPUTE GEMM` operations. These idle phases were marked for evaluation.

Simulation results indicated that the total VTA operation duration, including memory and non-memory tasks, spanned 14,022,760 cycles. Of this, memory-related operations accounted for 6,694,660 cycles, with actual memory access requiring 4,065,772 cycles. The remaining cycles were identified as cumulative idle periods,

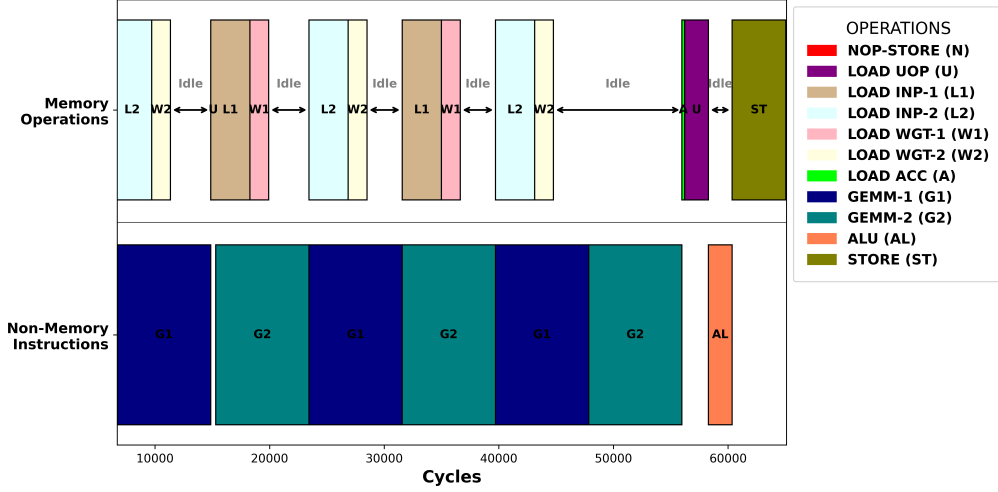


Figure 5.6: Memory Latency Extracts of the First Layer.

suggesting that approximately:

$$\text{Memory Idle Phase Percentage} = \frac{(6,694,660 - 4,065,772) \text{ cycles}}{6,694,660 \text{ cycles}} \times 100$$

$$\text{Memory Idle Phase Percentage} = \frac{2,628,888 \text{ cycles}}{6,694,660 \text{ cycles}} \times 100 \approx 39.27\%$$

of the memory operation time was spent idle. By applying the prefetching process described in Algorithm 3, the analysis indicated that a significant portion of this 39.27% idle time could be utilised, potentially reducing overall memory execution time.

The identified idle phases were evaluated against the prefetching mechanism to trigger memory operations that met the following conditions:

- **Completion of Predecessor:** Prefetching was initiated only when the predecessor memory operation had been completed.
- **Absence of Unfulfilled Dependencies:** Prefetching was allowed only when there were no unfulfilled dependencies, ensuring that operations marked for prefetching did not cause conflicts or additional delays.
- **No Conflicting Pending Operations:** The operation to be prefetched had no direct relation or dependencies with other pending operations, either memory or non-memory.

These conditions ensured that prefetching proceeded without causing disruptions to the execution flow. When all conditions were met, data for subsequent memory operations was prefetched ahead of pending operations with unfulfilled dependencies. All operations meeting the prefetch criteria were reordered and executed early, as reflected in the optimised schedule.

5.3.1.2 Analysis of Prefetchable Operations

The analysis, based on the outlined conditions, reveals that LOAD WGT operations are the most suitable for prefetching, as they are less dependent on LOAD INP operations, which are often delayed by non-memory tasks. LOAD WGT operations occur

frequently, typically in pairs, and require shorter execution windows compared to LOAD INP operations.

The total number of LOAD WGT operations identified was 992, of which 433 were deemed suitable for prefetching. As shown in Table 5.4, most of these operations (348) are found in Layers 1-9, while the remaining operations are distributed across Layers 10-13. No prefetchable operations were detected in Layers 14-18.

Table 5.4: Prefetchable and Non-Prefetchable LOAD WGT Operations across Layer Groups

Layer Group	Total WGT Operations	LOAD Operations	Prefetchable Operations	Non-Prefetchable Operations
Layers 1-9	512		348	164
Layers 10-13	224		85	139
Layers 14-18	256		0	256
Total	992		433	559

The prefetchable LOAD WGT operations are reordered to allow for earlier execution, as illustrated in Figure 5.7, where green marks indicate the LOAD WGT being fetched earlier.

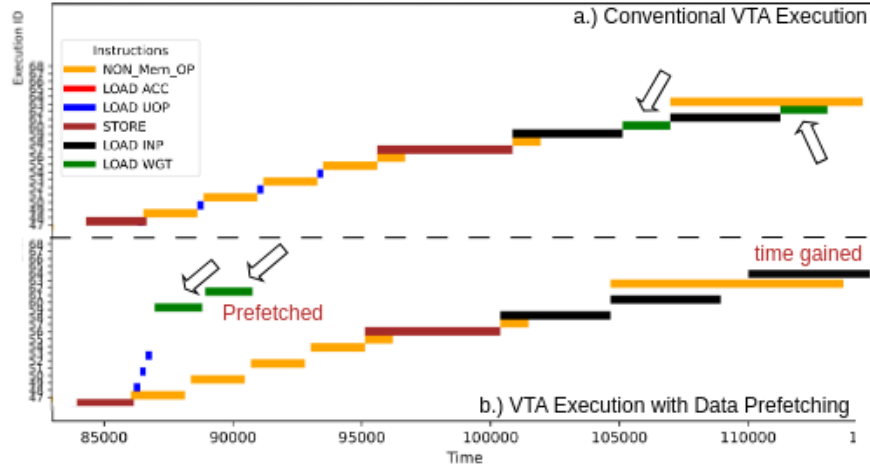


Figure 5.7: Prefetching Performance Evaluation.

Figure 5.7 shows the VTA simulation timeline before and after applying the data prefetching approach. The impact of prefetching across the entire VTA operation is evident, showing a reduction in memory idleness and a saving of approximately 12.68% of the overall execution time.

5.3.2 Computation and Evaluation of the Impact of Prefetching

The prefetching algorithm reorders 433 LOAD WGT operations to execute during idle periods caused by dependency delays in LOAD INP tasks. The pseudocode in Algorithm 11, alongside Equation 4.3, identified these 433 LOAD WGT operations as suitable for prefetching. The reduction in idle time, $T_{\text{prefetchedOP}}$, resulting from the prefetching approach by execution reordering is given by:

$$T_{\text{prefetchedOP}} = \sum_{j \in \text{Prefetched}} (t_j^{\text{finish}} - t_j^{\text{start}}) = (348 \times 1640) + (85 \times 3280) = 849,520 \text{ cycles}$$

This corresponds to approximately:

$$\frac{849,520 \text{ cycles}}{6,694,660 \text{ cycles}} \times 100 \approx 12.68\%$$

of the total VTA operation time saved by reducing idle periods.

Out of the total memory operation time of 6,694,660 cycles, idle phases accounted for 2,628,888 cycles. However, prefetching saved 849,520 cycles, constituting approximately 12.68% of the total memory operation duration. The memory idle phase, previously accounting for 39.27% of total memory operations, was reduced by around 32.32%.

This prefetching approach demonstrates a notable reduction in overall memory execution time. Figure 5.8 presents a plot comparing execution times before and after prefetching, showing that the prefetching algorithm reduced idle time by approximately 12.68%.

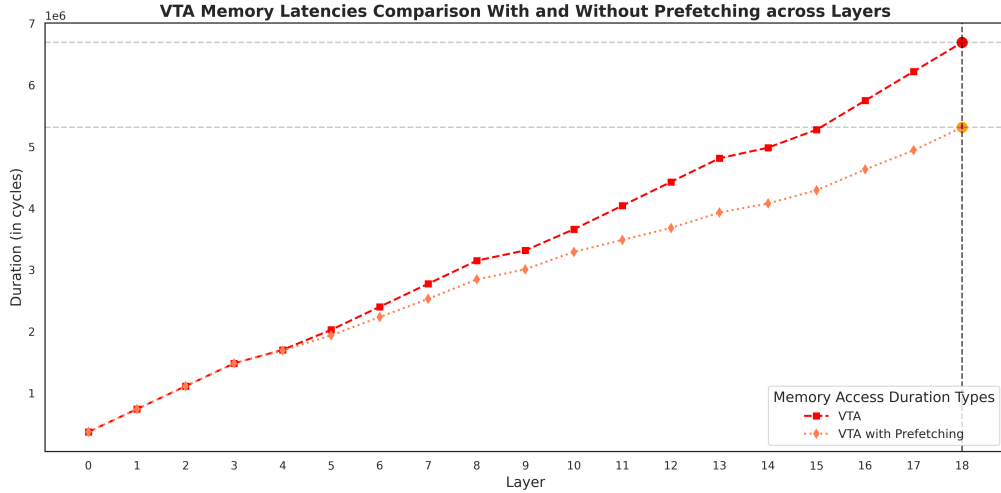


Figure 5.8: Memory Latency Comparison across Computational Layers of TTmaVTA and OPTTmaVTA with Prefetching.

The performance gains from prefetching are illustrated through a reduction in memory execution time, as shown in Figure 5.8, comparing memory access times between TTmaVTA and the optimised TTmaVTA with prefetching. The figure highlights improvements in memory access times across all layers, demonstrating the effectiveness of the prefetching strategy. In particular, the comparison between the original VTA and the OPTTmaVTA shows significant enhancements in execution time, memory access predictability, and memory throughput, especially for memory-bound tasks.

Further analysis of the performance improvements, as depicted in Figure 5.8, confirms the reduced idle time and increased memory efficiency achieved through

the prefetching algorithm. By addressing idle periods caused by unfulfilled dependencies, the prefetching of `LOAD WGT` operations alone reduces idle time by approximately 32.32%, leading to substantial improvements in memory utilisation and overall VTA performance.

Future research may focus on optimising the scheduling of non-memory operations, such as `GEMM`, to further reduce dependency delays and enhance overall system performance. Additionally, the caching mechanism evaluated in the following section presents further opportunities for improving memory throughput, indicating that additional strategies, such as caching, could enhance system efficiency further.

5.4 Intermediate Caching of Repetitive Memory Tasks

This section outlines the experimental setups, tests, and evaluations conducted to optimise VTA architecture’s memory efficiency and caching. The experiments are focused on evaluating intermediate caching strategies for repetitive memory tasks and assessing their impact on system performance. The test evaluations were carried out on the VTA architecture using a simulation environment extended to support caching mechanisms. In the VTA architecture, although intermediate memory is embedded within various modules to handle load inputs, weight micro-operations, and store outputs, a significant portion of intermediate data resides in external memory, contributing to increased latency. Analysis reveals that approximately 53.17% of the 126 *Store* operations written to the main memory were subsequently read by *load* operations, contributing to a notable portion of latency (8.8%) in main memory accesses.

5.4.1 Evaluation of Caching Optimisation

The caching optimisation methodology employed in Section 4.4 focuses on identifying frequently accessed memory regions defined as *write-read pairs* operations (*cache pairs*) for caching. The cache algorithm maps write memory tasks (*Store* operations) to corresponding read operations, forming *cache pairs*. These pairs are assigned unique cache identifiers (`CacheID`) and cache bits to facilitate cache management.

Algorithmically, a total of 339 memory access addresses experience repetitive reads, with 226 store-load pairs identified as suitable for caching. This quantity of *cache pairs* represents approximately 66.67% of the 339 cacheable memory operations. Among these, 32 store operations (25.4%) are identified as valid candidates for caching, while 194 corresponding read operations (19.5%) access data directly from the cache, significantly reducing memory access latency.

The proposed intermediate caching strategy demonstrates improved performance, particularly for repetitive memory tasks. For instance, 291 out of 992 load input (`LOAD INP`) operations (29.33%) retrieve data from the cache, reducing the number of memory accesses and freeing the memory bus for other operations. These results indicate that introducing intermediate caching for such repetitive tasks significantly enhances memory throughput and reduces latency.

5.4.1.1 Optimal Caching Threshold

Caching effectiveness depends on the frequency of access and the data size. Equation 4.12 calculates the optimal caching threshold, Th_{opt} , by considering several factors, including access frequency, data size, and bus utilisation, relative to the number of reads.

$$Th_{\text{opt}} = \frac{\text{cacheable_reads}}{\text{total_reads}} \times 100$$

By applying Equation 4.12, the optimal threshold, Th_{opt} , filters out non-effective caching operations. This threshold is computed as 22.77%, representing the ratio of read operations that can be retrieved from the cache before the next write operation. Thus, the optimal threshold, Th_{opt} , allows only 226 operations that meet the cache pairing benchmark. Of these cacheable operations, 32 are store operations, and 194 are load operations, establishing valid *write-read cache pairs*. Figure 5.9 shows a valid *cache pair* with a single write and multiple reads.

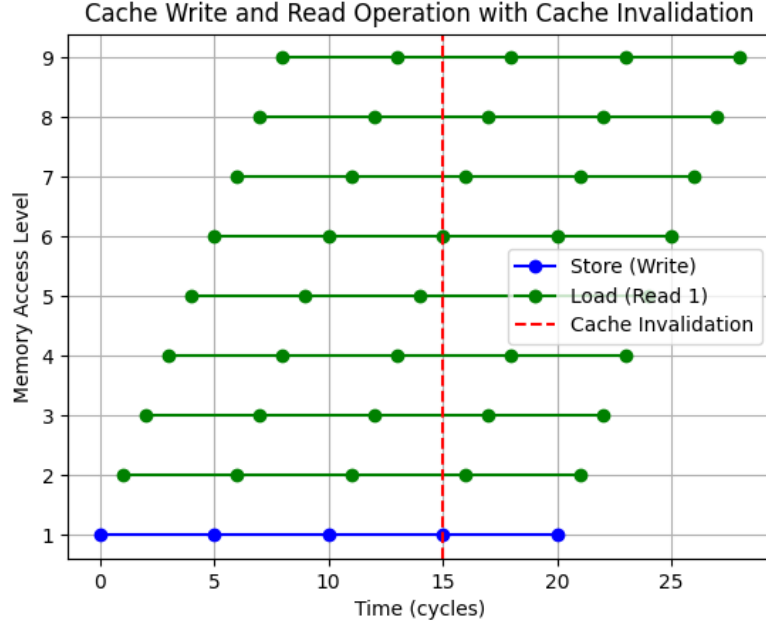


Figure 5.9: Latency of a single write with multiple read accesses to the same memory address.

Figure 5.9 also illustrates the latency experienced by repetitive memory accesses, where multiple subsequent read operations read a single store operation, each incurring memory access overhead.

5.4.1.2 Cache Resource Management and Address Allocation

For efficient cache management, the `cache pairs` identified by the caching algorithm are assigned a `CacheID` and allocated cache size while dynamically updating the base addresses of each cache category. Each pair's `CacheID` is mapped to a cache address starting from the base address, `0xA0000000`. The base address for

each category of **cache pairs** is determined by calculating the total byte size required for each operation, including the **x_size**, **y_size**, and strides. The cache depth is calculated based on the memory block size, data size and burst transfer rates. Computing each cache category cache address from Equation 4.14 ensures accurate memory allocation and avoids cache fragmentation.

$$\text{Cache_Address}_{n+1} = \text{Base_Address}_n + C_{\text{depth}}(n) \quad (5.5)$$

Here, Base_Address_n is the cache address for the current **CacheID**, and $C_{\text{depth}}(n)$ is the cache depth of the current **CacheID**, derived from the memory allocation of its corresponding **cache pairs**.

Each subsequent **cache pair** address is computed with an offset based on the depth of the previous total byte size. This ensures effective memory allocation and prevents address conflicts. By updating the cache table dynamically, the algorithm reduces memory access latency. Completed **cache pairs** memory regions are reallocated to new **cache pairs** when conditions allow.

The memory depth for each category is computed based on the cumulative size of the write operations (*Store*) of that category, ensuring that each **cache pair** is allocated a unique memory region. The caching algorithm dynamically updates the base addresses to accommodate all **cache pairs** as they are identified.

The test-enabled VTA simulator incorporates cache management features, including support for *cache bits* and dynamic address allocation. Operations with **CacheID** are marked with caching flags using a binary *cache bits*, where ‘1’ indicates a cache operation, and ‘0’ indicates a standard memory access. The identified *LOAD INP* operations flagged with a ‘1’ utilise the cached data. This cache-enhanced test-enabled simulator shows improved throughput by reducing memory access latency and alleviating the burden on the main memory.

5.4.1.3 Load Input Operations: Structure, Cache Access Latencies, and Burst Rates

To provide a structured overview of the memory load operations and their respective computation requirements, the number of cycles for each operation type across different layers, along with caching and memory access parameters, is detailed to aid in performing the necessary computations.

For the 194 total *cache pairs* *LOAD INP* operations, the cycle counts across layers are summarised in Table 5.5.

Table 5.5: *LOAD INP* Operational Statistics

Layer Group	No. of Operations (N)	Memory Cycles (T _{mem})	Total Memory Cycles (T _{mem} * N)	Cache Cycles	Burst Transfer Parameters
Layers 1-10	80	4,185 cycles	334,800 cycles	15 cycles	stride = 9, x_size = 56, y_size = 56
Layers 11-15	102	2,380 cycles	245,140 cycles	15 cycles	stride = 14, x_size = 14, y_size = 14
Layers 16-18	12	945 cycles	11,340 cycles	15 cycles	stride = 7, x_size = 7, y_size = 7
Total	194		591,280 cycles		

The table provides a breakdown of the memory access times ($T_{\text{read_mem}}$) and cache access times ($T_{\text{read_cache}}$) for each of the layer groupings, considering the burst rates. Each cache access's burst transfer rates based on stride, x_{size} , and y_{size} are utilised layer-wise to fetch data.

5.4.1.4 Cache Depth Sizing and Cache Address Computation

The cache depth, C_{depth} , for each *write-read pair* can be calculated using the burst transfer parameters (stride, x_{size} , and y_{size}). Table 5.6 outlines the burst transfer parameters and memory addresses for the traced write operations that form the *write-read cache pairs*.

ID	Layer	Latency	OpCode	Address	y _{size}	x _{size}	Stride	Freq
1	layer_0	4680	STORE	0x06090000	448	3136	2	3
2	layer_0	4680	STORE	0x06091880	448	3136	2	2
3	layer_2	4320	STORE	0x06090000	448	3136	2	3
4	layer_2	4320	STORE	0x06091880	448	3136	2	2
5	layer_5	4320	STORE	0x0600e000	196	784	4	2
6	layer_5	4320	STORE	0x0600ec40	196	784	4	1
7	layer_5	4320	STORE	0x0600e0c4	196	784	4	1
8	layer_5	4120	STORE	0x0600e188	196	784	4	1
9	layer_5	4120	STORE	0x0600e24c	196	784	4	1
10	layer_7	4120	STORE	0x06080000	392	784	2	2
11	layer_7	4120	STORE	0x06080188	392	784	2	1
12	layer_7	4020	STORE	0x06080620	392	784	2	1
13	layer_7	4020	STORE	0x060807a8	392	784	2	1
14	layer_7	4020	STORE	0x06080c40	392	784	2	1
15	layer_7	4020	STORE	0x06080dc8	392	784	2	1
16	layer_7	4020	STORE	0x06081260	392	784	2	1
17	layer_7	4020	STORE	0x060813e8	392	784	2	1
18	layer_8	4680	STORE	0x06090188	392	784	2	1
19	layer_8	4020	STORE	0x060907a8	392	784	2	1
20	layer_10	4120	STORE	0x06090000	784	784	1	3
21	layer_10	4020	STORE	0x06090310	784	784	1	1
22	layer_10	4680	STORE	0x06090620	784	784	1	1
23	layer_10	4120	STORE	0x06090930	784	784	1	1
24	layer_12	4120	STORE	0x0600e000	784	784	1	2
25	layer_12	4020	STORE	0x0600e310	784	784	1	1
26	layer_12	4120	STORE	0x0600e620	784	784	1	1
27	layer_12	4020	STORE	0x0600e930	784	784	1	1
28	layer_13	4020	STORE	0x06080000	784	784	1	2
29	layer_15	4020	STORE	0x0600d800	784	784	1	1
30	layer_15	4020	STORE	0x0600db10	784	784	1	1
31	layer_17	4020	STORE	0x05fcd000	784	784	1	1
32	layer_17	4020	STORE	0x05fcd310	784	784	1	1

Table 5.6: *STORE* Operations *Cache Pair* Statistical Structure.

The cache base address $C_{\text{base}}^{(n+1)}$ for the next *cache pair* can be determined using the formula:

$$C_{\text{base}}^{(n+1)} = C_{\text{base}}^{(n)} + C_{\text{depth}}$$

Where:

- $D_{\text{size}} = 128\text{-bit}$ (cache width).
- $A_{\text{width}} = 32\text{-bit}$ (address width).
- S_{stride} is the burst stride.
- $X_{\text{size}}, Y_{\text{size}}$ are the x and y dimensions.

The cache depth (C_{depth}) depends on the burst transfer rates for each layer. Using Table 5.6, the total write bytes can be calculated and converted into a cache size. The formula for cache depth is given as:

$$C_{\text{depth}} = \text{stride} \times x_{\text{size}} \times y_{\text{size}}$$

For the write operation in Layer 1-4 (L1):

$$C_{\text{depth}} = 2 \times 3136 \times 448 = 2,809,088 \text{ bytes}$$

For the write operation in Layer 5 (L2):

$$C_{\text{depth}} = 4 \times 784 \times 196 = 614,656 \text{ bytes}$$

For the write operation in Layer 7-8 (L3):

$$C_{\text{depth}} = 2 \times 784 \times 392 = 614,656 \text{ bytes}$$

For the write operation in Layer 9-18 (L4):

$$C_{\text{depth}} = 1 \times 784 \times 784 = 614,656 \text{ bytes}$$

Thus, the total cache depth size in bytes will be the sum of all layers, calculated as:

$$C_{\text{total}} = C_{\text{depth}}^{(L1)} \times 4 + C_{\text{depth}}^{(L2)} \times 5 + C_{\text{depth}}^{(L3)} \times 10 + C_{\text{depth}}^{(L4)} \times 13$$

$$C_{\text{total}} = 2,809,088 \times 4 + 614,656 \times 5 + 614,656 \times 10 + 614,656 \times 13$$

$$C_{\text{total}} = 11,236,352 + 3,073,280 + 6,146,560 + 7,990,528 = 28,446,720 \text{ bytes}$$

The total cache depth is approximately 28.45 MB.

The cache size can be computed for a 128-bit width (16 bytes per cache entry). The total cache size in bytes is calculated as:

$$\text{Total Cache Size} = \frac{C_{\text{total}}}{1024} = \frac{28,446,720}{1024} \approx 27,781 \text{ B}$$

Thus, summing the computed cache depths from Table 5.6 and applying the appropriate conversions, the total cache size is approximately 27.15 KB.

5.4.1.5 Optimisation of Cache Invalidation and Reallocation

The total cache size can be calculated based on the burst transfer information and cache depth values. However, maximising resource utilisation is critical in memory access. The cache depth is enhanced by allowing cache invalidation to reuse previously allocated cache regions under conditions of multiple writes, as observed in address 0x06090000 in Figure 5.10. Cache block invalidation allows efficient reuse of cache regions, reducing overall memory resource consumption.

Figure 5.10 represents a typical scenario where a single write operation (**Store_3**), to address 0x06090000 has eight corresponding read operations (**Load_3a** to **Load_3h**)

across layers 11 and 12, increasing the clock cycles utilised for accessing the main memory. Such operations occupy eight load cycles, consuming memory bus resources and increasing memory access latencies. This scenario highlights the need to optimise memory access patterns. Caching frequently accessed data enables lower latency access, as such frequently accessed data can be retrieved from the cache, significantly reducing latency, minimising the load on main memory, and improving memory throughput.

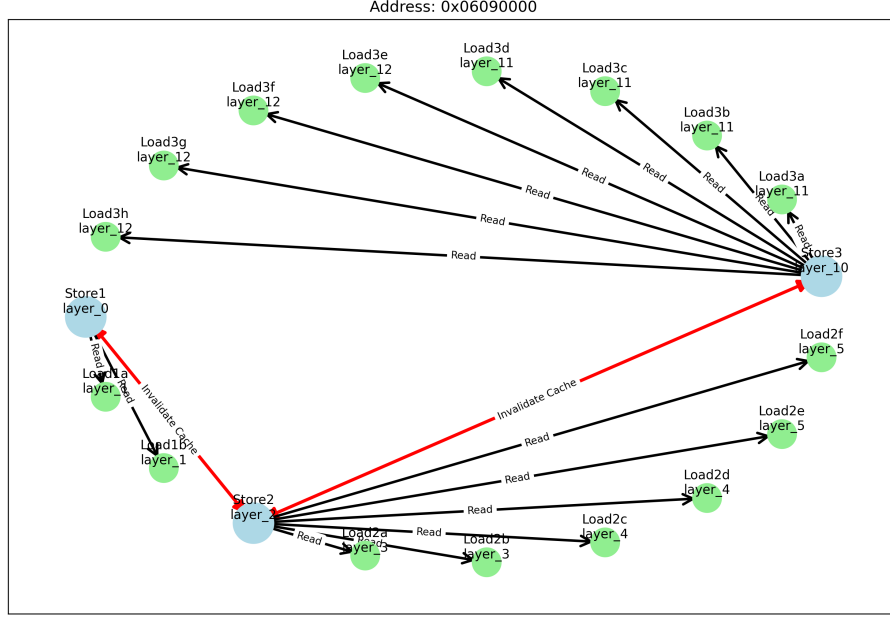


Figure 5.10: Graph of multiple writes and reads to address 0x06090000.

The VTA baseline graph in Figure 5.10 also illustrates multiple **STORE** and **LOAD** INP operations interacting with the same memory address (0x06090000). The plot highlights multiple writes to the same memory in red arrows and reads from the same memory region in black arrows. The cache for such repeated writes and reads can be reused. To reuse the cache due to multiple writes, the plot marks data overwrites indicated by red arrows and demonstrates the benefits of cache invalidation optimisation. Efficient invalidation conserves cache resources and reduces fragmentation. Repeated cache allocation for multiple writes to the same address also leads to inefficient cache utilisation. An optimised strategy deployed in this dissertation invalidates previous cache blocks on overwrite, allowing new data to be written without allocating separate blocks, thus conserving cache resources and improving efficiency.

To account for cache reuse, the total cache size of approximately 27.78 KB can be reduced by deducting the cache sizes of **CacheIDs** 3, 4, 20, 24, and 28 from Table 5.6. The initial reusable cache regions based on observed rewrites are calculated as follows:

$$\text{Reusable } C_{\text{depth}} = \text{stride} \times x_{\text{size}} \times y_{\text{size}}$$

For write operations with **CacheIDs** 3 and 4 (Ra1):

$$\text{Reusable } C_{\text{depth}} = 2 \times 3136 \times 448 = 2,809,088 \text{ bytes}$$

For write operations with **CacheIDs** 20, 24, and 28 (Ra2):

$$\text{Reusable } C_{\text{depth}} = 1 \times 784 \times 784 = 614,656 \text{ bytes}$$

Thus, the reusable cache depth size in bytes will be the sum of all layers, calculated as:

$$\text{Reusable } C_{\text{Summation}} = \text{Reusable } C_{\text{depth}}^{(Ra1)} \times 2 + \text{Reusable } C_{\text{depth}}^{(Ra2)} \times 3$$

$$\text{Reusable } C_{P1\text{total}} = 2,809,088 \times 2 + 614,656 \times 3$$

$$\text{Reusable } C_{P1\text{total}} = 5,618,176 + 1,843,968 = 7,462,144 \text{ bytes}$$

The part 1 reusable cache depth is approximately 7.46 MB.

With this approach, the store nodes (shown in blue ovals of Figure 5.10) represent new data written to the same address, while the previously stored data is invalidated, allowing subsequent store operations to overwrite the previously stored data (indicated by red arrows). This limits avoidable unnecessary cache allocation and maximises available resources.

Furthermore, earlier written data in the cache becomes obsolete once the reads are completed in earlier layers, making the corresponding cache regions redundant. These redundant regions can be freed and reused to optimise resource usage. Analysis shows that out of the 32 unique store cache cases, apart from the earlier listed nine repeated write memory access addresses which can be cached, 14 previously allocated caches have their reads completed at various intervals, allowing subsequent *STOREs* to invalidate and reuse those cache regions.

The total cache depth can be determined using information from Table 5.6. The actual cache depth is the difference between the total cache depth (comprising **CacheID** 1–32 from Table 5.6) and the invalidated/reused cache depths from repeated write addresses with previously read cache regions. The fourteen **CacheIDs** that utilise reusable caches include 14, 16, 18, 19, 21, 22, 23, 25, 26, 27, 29, 30, 31, and 32. The reusable cache data is structured in Table 5.7, which lists only the **CacheIDs** with allocated caches with each invalidated and reallocated **CacheID**.

ID	Layer	Latency	OpCode	Address	ysize	xsize	Stride	Inv_IDs
1	layer_0	4680	STORE	0x06090000	448	3136	2	3, 20, 27
2	layer_0	4680	STORE	0x06091880	448	3136	2	4, 16
5	layer_5	4320	STORE	0x0600e000	196	784	4	14, 24
6	layer_5	4320	STORE	0x0600ec40	196	784	4	18, 30
7	layer_5	4320	STORE	0x0600e0c4	196	784	4	29
8	layer_5	4120	STORE	0x0600e188	196	784	4	31
9	layer_5	4120	STORE	0x0600e24c	196	784	4	32
10	layer_7	4120	STORE	0x06080000	392	784	2	19, 28
11	layer_7	4120	STORE	0x06080188	392	784	2	21
12	layer_7	4020	STORE	0x06080620	392	784	2	22
13	layer_7	4020	STORE	0x060807a8	392	784	2	23
15	layer_7	4020	STORE	0x06080dc8	392	784	2	25
17	layer_7	4020	STORE	0x060813e8	392	784	2	26

Table 5.7: Cache operations showing Cache Invalidations (Inv_IDs)

The **Inv_IDs** column indicates which **CacheIDs** have been replaced in previous invalidations. For example, **CacheID** 1 invalidated and replaced **CacheIDs** 3, 20, and 27.

From the data on Table 5.7, the cache invalidation for part 2 of the reusable cache depth is calculated as follows:

$$\text{Reusable } C_{\text{depth}} = \text{stride} \times x_{\text{size}} \times y_{\text{size}}$$

For write operations with **CacheIDs** 14, 16, 18, and 19 (Rb1):

$$\text{Reusable } C_{\text{depth}} = 2 \times 784 \times 392 = 614,656 \text{ bytes}$$

For write operations with **CacheIDs** 21, 22, 23, 25, 26, 27, 29, 30, 31, and 32 (Rb2):

$$\text{Reusable } C_{\text{depth}} = 1 \times 784 \times 784 = 614,656 \text{ bytes}$$

Thus, the reusable cache depth size in bytes is the sum of all layers, calculated as:

$$\text{Reusable } C_{\text{P2total}} = \text{Reusable } C_{\text{depth}}^{(Rb1)} \times 4 + \text{Reusable } C_{\text{depth}}^{(Rb2)} \times 10$$

$$\text{Reusable } C_{\text{P2total}} = 614,656 \times 4 + 614,656 \times 10$$

$$\text{Reusable } C_{\text{P2total}} = 2,458,624 + 6,146,560 = 8,605,184 \text{ bytes}$$

The part 2 reusable cache depth is approximately 8.61 MB.

Hence, the total reusable cache depth consisting of part 1 and part 2 is:

$$\text{Reusable } C_{\text{total}} = \text{Reusable } C_{\text{P1total}} + \text{Reusable } C_{\text{P2total}}$$

$$\text{Reusable } C_{\text{total}} = 7,462,144 + 8,605,184 = 16,067,328 \text{ bytes}$$

Thus, the actual total cache size after accounting for cache invalidation is:

$$C_{\text{Total}} = C_{\text{total}} - \text{Reusable } C_{\text{depth}}$$

$$C_{\text{total}} = 28,446,720 - 16,067,328 = 12,379,392 \text{ bytes}$$

These conditions maximize cache usage, saving 19 out of the 32 cache depths as depicted in Table 5.7. Instead of allocating separate cache blocks for all write operations, cache optimisation can invalidate previously allocated caches and rewrite new data to the same block, conserving cache resources. This approach reduces memory access overhead, improves efficiency, and minimises cache fragmentation. Figure 5.11 shows the caching size improvement under different invalidation conditions. The figure illustrates how progressively applying invalidation techniques optimises the required cache size.

Therefore, the actual required **Total Cache Size** is:

$$\text{Total Cache Size} = \frac{C_{\text{total}}}{1024} = \frac{12,379,392}{1024} \approx 12,089 \text{ B}$$

$$\text{Total Cache Size} = 11.81 \text{ KB}$$

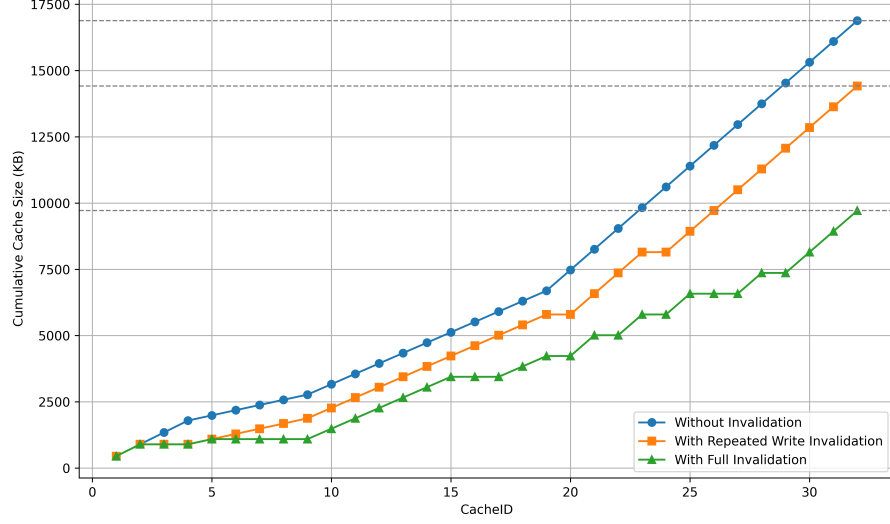


Figure 5.11: Cache Size Progression: No Invalidation vs. Invalidation Scenarios.

5.4.2 Performance Evaluation of Cache Optimisation

The performance evaluation includes computations related to the time saved by caching (T_{cacheOP}), the alternative formula for time saved by caching ($T_{\text{save_cache}}$), and the improvement in bus utilisation (B_{improve}).

5.4.2.1 Total Time Saved by Caching

The time saved by caching, T_{cacheOP} , is calculated as follows:

$$T_{\text{cacheOP}} = N_{\text{hits}} \times (T_{\text{mem}} - T_{\text{cache}}) \quad (5.6)$$

Where:

- T_{mem} is the time required for memory access:
 - 80 operations for layers 1-10, each taking 4,185 cycles.
 - 102 operations for layers 11-15, each taking 2,380 cycles.
 - 12 operations for layers 16-18, each taking 945 cycles.
- $T_{\text{cache}} = 15$ cycles for each cache access.
- $N_{\text{hits}} = 194$ operations in total.

The total time saved by caching is calculated by layer group as follows:

$$T_{\text{cacheOP}} = 80 \times (4185 - 15) + 102 \times (2380 - 15) + 12 \times (945 - 15)$$

$$T_{\text{cacheOP}} = 80 \times 4170 + 102 \times 2365 + 12 \times 930$$

$$T_{\text{cacheOP}} = 333,600 + 241,230 + 11,160 = 585,990 \text{ cycles saved by caching.}$$

An alternative formula to compute the time saved by caching ($T_{\text{save_cache}}$) uses the number of entries i as described in Equation 4.15:

$$T_{\text{save_cache}} = \sum_{i=1}^N (T_{\text{read_mem}}[i] - T_{\text{read_cache}}[i])$$

Where:

- $T_{\text{read_mem}}[i]$ represents the time for reading from the main memory.
- $T_{\text{read_cache}}[i]$ represents the time for reading from the cache.
- $N = 194$ total cacheable read operations.

Substituting the values for each group:

$$T_{\text{save_cache}} = (334,800 - 80 \times 15) + (245,140 - 102 \times 15) + (11,340 - 12 \times 15)$$

$$T_{\text{save_cache}} = (334,800 - 1,200) + (242,760 - 1,530) + (11,340 - 180)$$

$$T_{\text{save_cache}} = 333,600 + 241,230 + 11,160 = 585,990 \text{ cycles saved by caching.}$$

5.4.2.2 Bus Utilisation Improvement

The improvement in bus utilisation, B_{improve} , is computed using:

$$B_{\text{improve}} = \frac{T_{\text{save}}}{T_{\text{total_execution}}}$$

Where the total execution time without caching, $T_{\text{total_execution}}$, is 6,694,660 cycles. Substituting $T_{\text{save}} = 585,990$:

$$B_{\text{improve}} = \frac{585,990}{6,694,660} \approx 0.0875$$

Thus, the improvement in bus utilisation is approximately 8.75%.

The VTA simulator is adapted to simulate and evaluate the caching mechanism, modifying the input files to include cache bit and cache address information for each *cache pair* during OPTTmaVTA task execution in the test-enabled VTA Simulator. Tasks with a cache bit of '0' operated normally, accessing data from memory directly, while tasks with a cache bit of '1' utilised the cache for reading and writing data, freeing memory resources for subsequent tasks with a '0' cache bit. The evaluation results demonstrated significant improvements in memory access latency and overall system performance.

The final pseudocode of Algorithm 11 underwent rigorous testing and refinement to assess its effectiveness in caching optimisation of frequently accessed data. Figure 5.12 compares TTmaVTA without caching optimisation and optimised variant of the TTmaVTA with caching. In the optimised scenario, data from the (STORE) operations (S1 and S2), marked in brown, are stored in the cache at Layer 1, and this cached data is reused at Layer 2 by the (LOAD INP) operations (L1 and L2), marked in black. This configuration allows memory bus access for subsequent operations such as the load weights, enabling the early execution of four load weight operations (LOAD WGT), LW3, LW4, LW5 and LW6, concurrently with L1 and L2 within the same timeframe. This concurrent execution is possible because (LOAD INP) retrieves

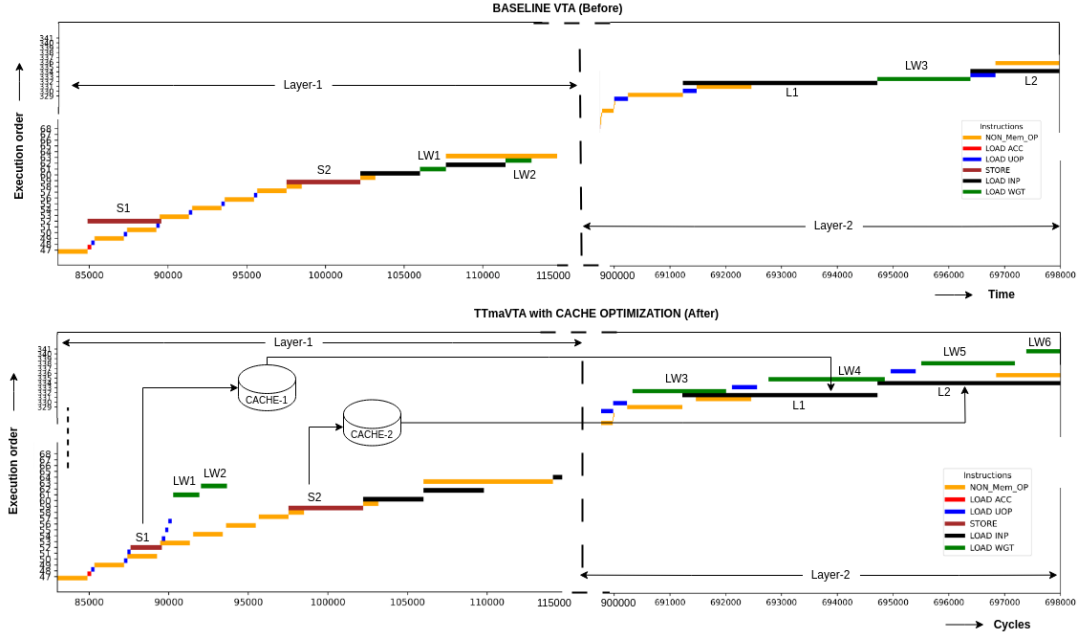


Figure 5.12: Caching Performance Evaluation.

data from the cache while (LOAD WGT) retrieves data from memory. This behaviour differs from the baseline VTA scenario, where only one load weight operation (LW3) could be executed in that interval.

Utilising data directly from the cache during load input (LOAD INP) operations eliminated the time typically spent accessing data from memory, leading to a **29.33%** reduction in LOAD INP memory access time. The caching methodology discussed and evaluated in this section ensured the successful implementation of the caching algorithm, resulting in a significant reduction in memory access latency.

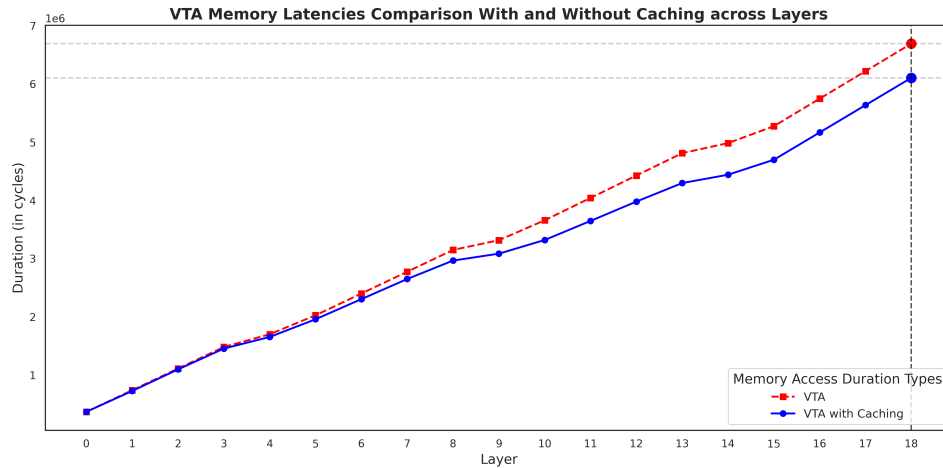


Figure 5.13: Memory Latency Comparison across Computational Layers of TTmaVTA and OPTTmaVTA with Caching.

Specifically, the optimised algorithm saved approximately **585,990 cycles** of the total TTmaVTA execution time. The simulated results showed an **8.75%** improvement in execution time over the baseline TTmaVTA architecture, as depicted in Figure 5.13. These results highlight the effectiveness of the memory write-read

operation *cache pair* algorithm in optimising cache usage, reducing memory access latency, and improving hardware accelerator efficiency. Applying the caching technique across the entire VTA operation while considering VTA baseline execution ordering, non-memory operations, and all dependencies resulted in a **29.33%** reduction in `LOAD INP` memory access time and an overall **8.75%** improvement in VTA execution time. Notably, applying intermediate caching to load input read operations with its dependencies demonstrated superior results, as this alleviates and eases the memory bus burden, allowing other memory operations to execute concurrently, thereby improving memory throughput.

5.5 Evaluating Comprehensive Enhancement of the Incorporated Optimisation Strategies

To fully capture the benefits of both optimisation approaches (prefetching and caching) across all memory operations, the time saved from all applied optimised versions can combine to reflect total time savings across all memory accesses.

The evaluation in Section 5.3 demonstrated the ability of the prefetching approach to perform execution during memory idle phases. In this referenced section, the prefetch algorithm was evaluated to assess its effectiveness in enhancing memory access efficiency and reducing latency caused by unfulfilled dependencies of operations during memory accesses. This evaluation involved vital activities such as extending and modifying the VTA simulator, identifying and analysing memory idle periods and testing the prefetch bit functionality.

To ensure optimal memory utilisation, the prefetching algorithm systematically identified gaps or idle periods between memory tasks and sorted and recorded the idle periods to quantify the duration of memory idle phases. The primary cause of memory idleness identified was dependency-related stalls by memory operations. For each idle period, the algorithm marked free time intervals and evaluated their eligibility for prefetching. The prefetch bit functionality is integrated into the simulator as a flag, which is evident in reordering memory access execution to proactively fetch data during idle periods for operations that meet the prefetching conditions. This mechanism, tested in simulation, validated its effectiveness in mitigating memory access latency, achieving a time gain of *12.68%* in memory access execution.

Furthermore, Section 5.4 focused on assessing the effectiveness of caching the memory *write-read cache pair* pseudocode of Algorithm 11 in improving memory access efficiency. Rigorous testing demonstrated the successful extension of the prefetch-ready test-enabled simulator to identify and tag *cache pairs* for caching and to optimise cache allocation utilisation. The dynamic allocation strategy and systematic tagging of `cache pairs` proved instrumental in achieving notable time performance improvements by *8.75%*. These incorporated prefetching and caching strategies significantly reduced memory access latency, enhancing overall system performance.

The evaluation in the following section confirmed the efficacy of the prefetching and caching strategies in improving memory access throughput by utilising substantial potential idle phases in VTA memory access for prefetching and also optimising cache utilisation for cacheable memory operations, thereby reducing memory latencies.

5.5.1 Evaluation of Incorporated Prefetching and Caching Strategies

Evaluating all optimisation strategies—memory contention resolution, prefetching, and caching, as discussed in Sections 5.3 and 5.4—facilitated a reduction in idle durations caused by stalled operations due to dependencies and reducing memory accesses to the main memory. By prefetching data during idle phases and caching frequently accessed data, the system harnessed previously unutilised memory access phases, thereby minimising idle time and improving memory throughput.

The reduction in idle periods after implementing the prefetching optimisation and reordering executions is given by quantifying the formula in Equation 4.3, which reflects the system’s idle periods after applying the prefetching strategy. Incorporating the prefetching and caching optimisations from the computations of Equations 4.3 and 4.15 in Sections 5.3.2 and 5.4.2, the total time saved for both strategies is calculated as follows:

$$T_{\text{total_saved}} = T_{\text{prefetchedOP}} + T_{\text{save_cache}}$$

$$T_{\text{save_cache}} = 585,990 \text{ cycles}$$

$$T_{\text{prefetchedOP}} = 849,520 \text{ cycles}$$

$$T_{\text{total_saved}} = T_{\text{prefetchedOP}} + T_{\text{save_cache}} = 849,520 + 585,990 = 1,435,510 \text{ cycles}$$

Without considering non-memory dependencies, the total memory access time improved by approximately *21.44%*, resulting in saving *1,435,510* cycles from the initial recorded total of 6,694,660 cycles of the TTmaVTA memory latencies.

5.5.2 Performance Results

The performance results of the following approaches are evaluated in the plot shown in Figure 5.14:

- **TTmaVTA Without Contention (WC):** Utilises a memory scheduling mechanism that addresses memory contention but introduces resource overhead.
- **OPTTmaVTA Without Contention, with Caching:** Reduces the overhead by caching frequently accessed data.
- **OPTTmaVTA Without Contention, with Prefetching:** Minimises memory latency by prefetching data during idle memory cycles.
- **OPTTmaVTA Without Contention, with Caching and Prefetching:** Combines both caching and prefetching techniques for optimal performance.

The saved time is directly reflected in reduced total memory access latency. Figure 5.14 demonstrates the impact of various memory optimisation techniques on latency reduction and memory access efficiency, including caching and prefetching.

To quantify the improvements introduced by the optimisations shown in Figure 5.14, each technique is evaluated as follows:

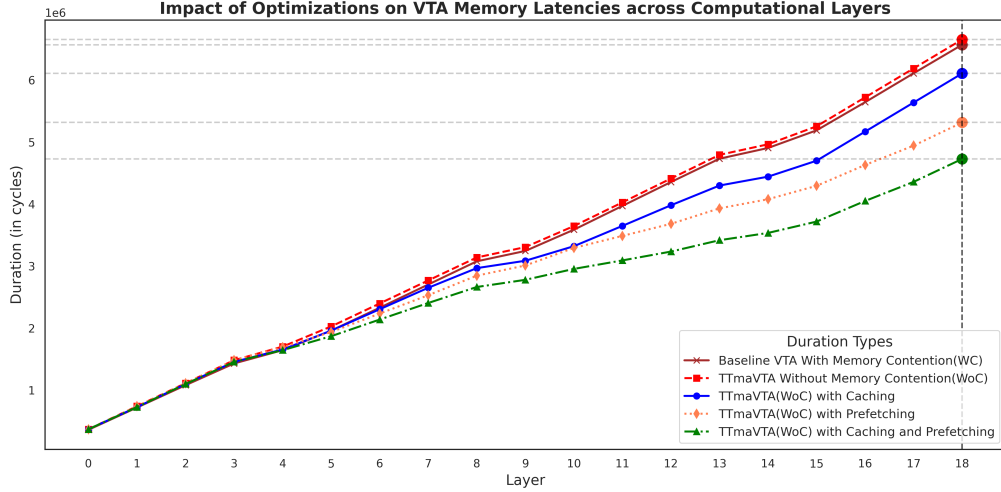


Figure 5.14: ResNet-18 Memory Latency Trend across Computational Layers for TTmaVTA and OPTTmaVTA.

- **TTmaVTA Without Optimisation:** TTmaVTA introduces a memory access scheduling mechanism that allows for scheduled memory accesses with timing guarantees, mitigating memory contention but slightly increasing memory latency. This results in an overhead of approximately **1.29%** due to the complexity of the memory access scheduling and increased resource utilisation.
- **TTmaVTA with Caching:** Incorporating caching into the TTmaVTA system reduces the overhead the memory scheduling mechanism introduces. Caching results in an **8.75% improvement** in memory latency compared to TTmaVTA without caching.
- **TTmaVTA with Prefetching:** Prefetching, which loads data into memory during idle cycles, provides more significant gains. It yields a **12.68% improvement** over TTmaVTA without optimisations.
- **TTmaVTA with Caching and Prefetching:** Combining caching and prefetching produces the highest performance, with a **19.86% improvement**, compared to TTmaVTA without optimisations, and it also achieves an additional **12.57% improvement** over caching alone.

By introducing caching and prefetching co-mechanisms, the OPTTmaVTA compensate for any overhead introduced by the memory scheduling system. The applied optimisations result in substantial performance improvements. Combining caching and prefetching is the most effective approach, reducing memory latency by approximately 20%.

The traditional VTA architecture exhibited significant memory contention, which adversely affected performance. In contrast, TTmaVTA introduced a structured memory access mechanism that alleviated memory contention and ensured timing guarantees. OPTTmaVTA further improved upon TTmaVTA by enhancing performance through caching and prefetching techniques at a reduced latency.

5.5.3 Absolute Time Reduction and Performance Improvement Computations (OPTTmaVTA)

The absolute time reduction due to the application of all optimised evaluations, which reflects the total reduction across all memory accesses, is denoted by A_{time} and is given by:

$$A_{\text{time}} = T_{\text{absolute}}^{(\text{no.opt})} - T_{\text{absolute}}^{(\text{opt})}$$

The absolute execution time without optimisation is:

$$T_{\text{absolute}}^{(\text{no.opt})} = 14,022,760 \text{ cycles}$$

For the recorded absolute execution time with optimisation:

$$T_{\text{absolute}}^{(\text{opt})} = 12,587,250 \text{ cycles}$$

The absolute time reduction is therefore:

$$A_{\text{time}} = 14,022,760 - 12,587,250 = 1,435,510 \text{ cycles}$$

Furthermore, the percentage of performance improvement due to the optimised computations of prefetching and caching, denoted as $P_{\text{improvement}}$, quantifies the effectiveness of the optimisation strategy. It is computed as:

$$P_{\text{improvement}} = \frac{A_{\text{time}}}{T_{\text{absolute}}^{(\text{no.opt})}} \times 100\%$$

Substituting the values:

$$P_{\text{improvement}} = \frac{1,435,510}{14,022,760} \times 100\% \approx 10.24\%$$

Thus, the performance improvement due to optimisation is approximately 10.24%. This percentage quantifies the improvement in execution time attributable to the integration of various optimisation techniques, reflecting the overall performance gains achieved. The percentage improvements for each optimisation technique are compared against the baseline as illustrated in Figure 5.14.

Finally, the memory access latencies evaluated using the test-enabled prefetch-cache-ready VTA software simulator, incorporating the various optimisation techniques, showed significant gains. The simulation compared the TTmaVTA and OPTTmaVTA with the conventional VTA in the context of ResNet-18. Applying the multi-stage memory access optimisations demonstrates a potential reduction of 10.24% in the total execution time of ResNet-18. These findings highlight both strategies' potential to advance hardware acceleration and improve system performance across computational tasks.

5.6 Limitations and Future Work

The proposed memory scheduling technique has the potential to be applied to various accelerators and architectures, offering a generalised framework for optimising

memory access in deep learning accelerators. However, it is essential to acknowledge that memory access patterns vary across various neural network structures. The findings in this dissertation focus specifically on ResNet-18, and applying the proposed technique to other neural networks would necessitate a thorough retracing step to capture another network’s unique memory access patterns and dynamics.

Moreover, although the work presented here is primarily centred on the VTA architecture, the methodology can be adapted to other accelerators. The approach can be modified to align with the specific architecture and design of the chosen accelerator, ensuring compatibility and optimal performance. While the schedules tested in the simulation environment have shown promising results, it is recommended that the simulated part be implemented in a hardware-based time-triggered dispatcher for further validation.

Additionally, extending this architecture to enable the scheduling of instructions across multiple pipelines presents significant opportunities for improving execution efficiency. It is projected that this could enhance performance by over 30%, given that approximately 70% of memory operations depend on non-memory instructions, which directly influence overall memory access latencies.

Conclusion

The increasing complexity of AI-driven applications, particularly in safety-critical and real-time systems, necessitates innovations in neural network accelerator designs to provide computational efficiency while guaranteeing predictability and timing determinism. This dissertation thoroughly investigates the challenges of memory access, predictability and optimisation in existing neural network accelerators, focusing on the Versatile Tensor Accelerator (VTA) related to safety-critical applications. The complexity of neural network structures, coupled with the demands for efficient memory performance, necessitates a comprehensive understanding of memory access patterns and their implications for hardware design. The dissertation highlights the significant issues related to memory access and access conflicts in resource-constrained environments such as FPGAs. A systematic analysis of existing methodologies shows that traditional approaches often neglect the system-level and architectural aspects crucial for ensuring reliable operation in safety-critical contexts. This gap in current research emphasises the need for advanced solutions that enhance performance while upholding safety and predictability.

This dissertation presents a comprehensive study of the VTA by using hardware-based memory tracing to estimate measurement metrics accurately and the VTA simulator, specifically focusing on its enhancements for simulating memory operations. Memory tracing allows for a detailed examination of memory behaviour during the execution of neural networks. The insights gained from this analysis identified performance bottlenecks and informed optimisation strategies within the VTA architecture. Several challenges, including the event-driven nature of VTA memory access handling, contention during memory accesses, memory idleness caused by unfulfilled dependencies, avoidable repetitive memory accesses, and execution overheads, were identified during memory access tracing of the VTA.

To address these challenges, this dissertation leverages the following innovative techniques:

- **Time-Triggered Memory Access:** This method enhances determinism by scheduling memory operations at fixed and known intervals, reducing variability in memory access times.
- **Contention Mitigation with Time-Based Schedules:** By managing all tasks, the time-triggered schedules mitigate access collisions and latency, ensuring timely memory resource allocation in all memory operations to enhance predictability.
- **Predictive Memory Management with Prefetching Mechanisms:** This method utilised prefetching to schedule subsequent operations when anticipated non-memory dependencies might cause a halt in the next memory op-

eration, optimising data loading and reducing idle phases, thus reducing the overall access time.

- **Data Caching with Deterministic Data Placement:** This technique identifies frequent memory accesses to organise and cache reusable data for subsequent reads to retrieve data from the cache, freeing the main memory. This method significantly enhanced system performance and reduced latency.

This dissertation presented the **TTmaVTA** (Time-Triggered Memory Access VTA), a novel hardware-based co-design enhancement of the VTA architecture, which addressed the inherent memory access limitations of conventional tensor accelerators in safety-critical environments. The TTmaVTA achieved predictable memory access without contention by employing a time-triggered framework orchestrated by the Time-Triggered Memory Access Controller (TTmaC). This controller operates by synchronising memory transactions with a global clock, pre-loading memory access schedules, and ensuring data is fetched at precise intervals ahead of the VTA modules' requests. The memory contention strategy effectively addressed the challenges associated with concurrent memory access attempts. By systematically aligning the timing of memory operations, the scheduling process ensures a more efficient utilisation of the memory bus. The memory access scheduling leverages memory trace data, ensuring that scheduled memory operations are conflict-free and temporally optimised.

The critical challenges associated with memory access in neural networks are effectively addressed through the scheduling algorithm developed, which leverages task order, dependencies, and memory access times, proving instrumental in ensuring predictable execution with guaranteed timing. This predictability is essential for high-performance neural network operations, enabling the execution of memory-bound tasks without memory contention. The subsequent comprehensive study of memory access patterns revealed opportunities for further optimising memory access in tensor accelerators. The focus lies on addressing memory-related inefficiencies, particularly those arising from operational dependency stalls, frequently avoidable bus accesses, and memory contention, which limit the performance of memory-bound computational tasks such as deep learning inference.

The hardware-based evaluation of the TTmaVTA framework, conducted on the Trenz Electronics TE0802 FPGA board using an optimised 2D Convolutional Neural Network (Conv2D) testbench as a workload, applied the time-triggered scheduling mechanism. Repeated runs of the model on TTmaVTA exhibited marginal improvements in mitigating memory access contention compared to the baseline VTA, with a 2.86% reduction in memory access time. However, this improvement came at the cost of increased system complexity and resource consumption overheads, which limit the scalability of hardware-based tracing for larger or more complex workloads.

Furthermore, to improve memory throughput, **OPTTmaVTA** is introduced, a novel approach to enhancing VTA with Time-Triggered Memory Access Optimisation. OPTTmaVTA is designed systematically to integrate advanced memory optimisation techniques, namely prefetching and caching. The prefetching mechanism, designed for intelligent data retrieval, exploited idle periods in memory operation timelines by anticipating and retrieving data during idle periods, thereby reducing latencies caused by memory access stalls due to unfulfilled dependencies. This strategic approach not only significantly reduces memory access latency but also

improves the overall efficiency of the VTA.

Moreover, the caching strategy, which stores and reuses frequently accessed data in write-read operation pairs, is crucial in optimising memory throughput and further reducing latency. By providing low-latency data retrieval from the cache, OPTTmaVTA significantly reduces the need for frequent access to the main memory, thus freeing up the memory bus for other operations. The caching methods enhance data retrieval speeds by managing cache entries through a robust eviction strategy and optimising the use of the memory bus, which is crucial for meeting the demands of contemporary neural network applications.

Experimentally, these strategies demonstrate the importance of proactive memory management in improving overall performance. The integration of these techniques yielded predictable memory access, improvements in memory performance and reliability of neural network accelerators, thereby meeting the timing requirements of safety-critical applications. The VTA simulator, modified and extended to support these optimisations, is utilised to export time-based schedules. With its extensions, the VTA simulator enabled cache-prefetch entries that allow for a detailed evaluation of the effectiveness of the integrated approaches. These evaluations demonstrated time-triggered memory access without collisions and the application of data prefetching by organising entries to optimise execution order. Additionally, the caching methods free up the memory bus, significantly enhancing the performance of the OPTTmaVTA and, by extension, the underlying memory access of hardware.

The iterative design process employed in this dissertation, using the test-enabled software simulator, proved effective in validating the applied optimisation strategies before hardware synthesis, ensuring that the final architecture aligns with performance expectations. Simulation-based testing demonstrated that prefetching reduced memory access latency by 12.68% while caching further contributed to a reduction by 8.75%. The combined effects of these optimisations resulted in a 21.44% reduction in memory access latency, corresponding to saving 1,435,510 cycles.

When assessing the impact of these optimisations on overall system performance with deep learning workloads, the OPTTmaVTA architecture achieved a total execution time reduction of 10.24% for ResNet-18, with an execution time of 28.736 ms compared to the conventional VTA simulator, which recorded an average of 32.014 ms per run. This performance improvement underscores the significance of memory optimisation strategies in enhancing the efficiency of hardware accelerators.

Moreover, the proposed memory scheduling techniques can be applied to other tensor accelerators and architectures, providing a generalised framework for optimising memory access in deep learning accelerators. However, memory access patterns vary across different neural networks. While this dissertation focuses on ResNet-18, applying these techniques to other neural networks would require retracing to capture the network’s specific memory access patterns and dynamics. Although the primary explored tensor accelerator dynamics centres on the VTA architecture, the same approach can be adapted for other accelerators by tailoring the methodology to align with their architectural design. While the generated schedules evaluated on the simulator performed optimally, hardware implementation with a time-triggered dispatcher is recommended. Furthermore, extending the architecture to scheduling instructions with multiple pipelines has the potential to improve execution by over 30%, given that approximately 70% of memory operations depend on non-memory

instructions, impacting total memory access latencies.

In conclusion, the findings of this dissertation contribute to the ongoing development of more efficient hardware acceleration frameworks by addressing key memory access bottlenecks. The proposed multi-stage optimisation approach—combining prefetching and caching—demonstrates the potential for substantial performance gains in memory-bound operations, thereby enhancing the overall computational efficiency of deep learning workloads and other applications that rely heavily on memory access. This dissertation establishes a pathway for advancing memory access strategies in existing neural network hardware accelerators such as the VTA, addressing the critical issues of predictability and performance in safety-critical environments. Future work may explore further refinements to these techniques and their applicability to other hardware platforms and emerging computational models. The proposed methodologies and their evaluation pave the way for future research, promoting the development of more robust and reliable neural network systems.

The work presented in this dissertation contributes to a significant advancement in the field of neural network accelerators by demonstrating that a time-triggered memory access architecture can substantially improve memory predictability, meeting the requirements for safety-critical applications. By combining prefetching and caching optimisation techniques, this architecture reduces memory access latencies and improves resource efficiency in FPGA-based accelerators. However, there remain opportunities for future research, particularly in addressing the variability introduced by event-triggered non-memory instructions and further optimising the interaction between memory and non-memory tasks with an integrated schedule. The experimental design provides a promising framework for enhancing the timing performance guarantees of neural network accelerators, bridging the gap between the flexibility of accelerators like VTA and the stringent requirements of safety-critical systems. As AI continues to evolve, the principles and methodologies established in this work will provide a foundation for future innovations in accelerator design, ensuring that computational memory throughput and temporal predictability can coexist in next-generation AI systems.

Bibliography

- [1] Sidharta Andalam et al. “A Predictable Framework for Safety-Critical Embedded Systems”. In: *IEEE Transactions on Computers* 63.7 (2014), pp. 1600–1612. DOI: 10.1109/TC.2013.28.
- [2] Aniebiet Micheal Ezekiel, Daniel Onwuchekwa, and Roman Obermaisser. “Optimization of the Versatile Tensor Accelerator (VTA) Load Module in a Time-Triggered Memory Access”. In: *2023 26th Euromicro Conference on Digital System Design (DSD)*. 2023, pp. 146–152. DOI: 10.1109/DSD60849.2023.00030.
- [3] Waleed M. Gondal et al. “Weakly-supervised localization of diabetic retinopathy lesions in retinal fundus images”. In: *2017 IEEE International Conference on Image Processing (ICIP)*. 2017, pp. 2069–2073. DOI: 10.1109/ICIP.2017.8296646.
- [4] Sergi Alcaide et al. “High-Integrity GPU Designs for Critical Real-Time Automotive Systems”. In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019, pp. 824–829. DOI: 10.23919/DATE.2019.8715177.
- [5] Jyotika Athavale et al. “AI and Reliability Trends in Safety-Critical Autonomous Systems on Ground and Air”. In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2020, pp. 74–77. DOI: 10.1109/DSN-W50199.2020.00024.
- [6] Caleb Adams et al. “Towards an Integrated GPU Accelerated SoC as a Flight Computer for Small Satellites”. In: *2019 IEEE Aerospace Conference*. 2019, pp. 1–7. DOI: 10.1109/AERO.2019.8741765.
- [7] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. pmlr. 2015, pp. 448–456.
- [8] Hamidreza Ahmadian, Roman Obermaisser, and Jon Perez. *Distributed Real-Time Architecture for Mixed-Criticality Systems*. CRC Press, 2018.
- [9] Ziyi Lv and Jing Zhang. “A Survey of FPGA-Based Deep Learning Acceleration Research”. In: *The International Conference on Image, Vision and Intelligent Systems (ICIVIS 2021)*. Ed. by Jian Yao et al. Singapore: Springer Nature Singapore, 2022, pp. 59–65. DOI: 10.1007/978-981-16-6963-7_5.
- [10] Stylianos Venieris, Alexandros Kouris, and Christos Bouganis. “Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions”. In: *ACM Computing Surveys* 51 (Mar. 2018). DOI: 10.1145/3186332.

- [11] Paolo Rech. “Artificial Neural Networks for Space and Safety-Critical Applications: Reliability Issues and Potential Solutions”. In: *IEEE Transactions on Nuclear Science* 71.4 (2024), pp. 377–404. DOI: 10.1109/TNS.2024.3349956.
- [12] Sanjoy Baruah, Alan Burns, and David Griffin. “Functional Uncertainty in Real-Time Safety-Critical Systems”. In: *Proceedings of the 30th International Conference on Real-Time Networks and Systems (RTNS 2022)*. RTNS. Paris, France: Association for Computing Machinery, 2022, pp. 1–11. ISBN: 9781450396509. DOI: 10.1145/3534879.3534884. URL: <https://doi.org/10.1145/3534879.3534884>.
- [13] John Stankovic and Krithivasan Ramamritham. “EDITORIAL What is Predictability for Real-Time Systems?” In: *Real-Time Systems* 2 (Oct. 1993).
- [14] Alessandro Biondi et al. “A Safe, Secure, and Predictable Software Architecture for Deep Learning in Safety-Critical Systems”. In: *IEEE Embedded Systems Letters* PP (Nov. 2019), pp. 1–1. DOI: 10.1109/LES.2019.2953253.
- [15] Ryan Luna and Sheikh Islam. “Security and Reliability of Safety-Critical RTOS”. In: *SN Computer Science* 2 (Sept. 2021). DOI: 10.1007/s42979-021-00753-y.
- [16] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D. Lawrence. “Challenges in Deploying Machine Learning: A Survey of Case Studies”. In: *ACM Comput. Surv.* 55.6 (Dec. 2022). ISSN: 0360-0300. DOI: 10.1145/3533378. URL: <https://doi.org/10.1145/3533378>.
- [17] Leeor Peled, Uri Weiser, and Yoav Etsion. “A Neural Network Prefetcher for Arbitrary Memory Access Patterns”. In: *ACM Trans. Archit. Code Optim.* 16.4 (Oct. 2019). ISSN: 1544-3566. DOI: 10.1145/3345000. URL: <https://doi.org/10.1145/3345000>.
- [18] Milad Hashemi et al. “Learning Memory Access Patterns”. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, July 2018, pp. 1919–1928. URL: <https://proceedings.mlr.press/v80/hashemi18a.html>.
- [19] Jie Sun et al. “Hardware resilient deep learning for healthcare”. In: *IEEE Transactions on Industrial Informatics* 17.4 (2021), pp. 2964–2973.
- [20] Jaume Abella et al. “SAFEXPLAIN: Safe and Explainable Critical Embedded Systems Based on AI”. In: *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10137128.
- [21] Cesar Torres-Huitzil and Bernard Girau. “Fault and Error Tolerance in Neural Networks: A Review”. In: *IEEE Access* 5 (2017), pp. 17322–17341. DOI: 10.1109/ACCESS.2017.2742698.
- [22] Lucas Matana Luza et al. “A Model-Based Framework to Assess the Reliability of Safety-Critical Applications”. In: Apr. 2021, pp. 41–44. DOI: 10.1109/DDECS52668.2021.9417059.

- [23] Zhiheng Zhong et al. “Machine Learning-Based Orchestration of Containers: A Taxonomy and Future Directions”. In: *ACM Comput. Surv.* 54.10s (Sept. 2022). ISSN: 0360-0300. DOI: 10.1145/3510415. URL: <https://doi.org/10.1145/3510415>.
- [24] Haitong Wang, Neil C. Audsley, and Wanli Chang. “Addressing Resource Contention and Timing Predictability for Multi-Core Architectures with Shared Memory Interconnects”. In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020, pp. 70–81. DOI: 10.1109/RTAS48715.2020.00–16.
- [25] Dakshina Dasari et al. “Identifying the sources of unpredictability in COTS-based multicore systems”. In: *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2013, pp. 39–48. DOI: 10.1109/SIES.2013.6601469.
- [26] Gianluca Brilli et al. “Evaluating Controlled Memory Request Injection for Efficient Bandwidth Utilization and Predictable Execution in Heterogeneous SoCs”. In: *ACM Trans. Embed. Comput. Syst.* 22.1 (Dec. 2022). ISSN: 1539-9087. DOI: 10.1145/3548773. URL: <https://doi.org/10.1145/3548773>.
- [27] Roman Obermaisser et al. “Adaptive Time-Triggered Multi-Core Architecture”. In: *Designs* 3 (Jan. 2019), p. 7. DOI: 10.3390/designs3010007.
- [28] Nithin Ravani Nanjundaswamy et al. “RISC-V Timing-Instructions for Open Time-Triggered Architectures”. In: *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2023, pp. 211–214. DOI: 10.1109/DSN-W58399.2023.00058.
- [29] Mario Qosja, Simon Meckel, and Roman Obermaisser. “Simulator for Time-Triggered Organic Computing”. In: *Procedia Computer Science* 220 (2023). The 14th International Conference on Ambient Systems, Networks and Technologies Networks (ANT) and The 6th International Conference on Emerging Data and Industry 4.0 (EDI40), pp. 127–134. ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2023.03.019>. URL: <https://www.sciencedirect.com/science/article/pii/S1877050923005549>.
- [30] Gautam Gala, Javier Castillo Rivera, and Gerhard Fohler. “Work-in-Progress: Cloud Computing for Time-Triggered Safety-Critical Systems”. In: *2021 IEEE Real-Time Systems Symposium (RTSS)*. 2021, pp. 516–519. DOI: 10.1109/RTSS52674.2021.00054.
- [31] Zachary Kenton et al. *Generalizing from a few environments in safety-critical reinforcement learning*. 2019. arXiv: 1907.01475 [cs.LG]. URL: <https://arxiv.org/abs/1907.01475>.
- [32] Jude Haris et al. “SECDA: Efficient Hardware/Software Co-Design of FPGA-based DNN Accelerators for Edge Inference”. In: *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2021, pp. 33–43. DOI: 10.1109/SBAC-PAD53543.2021.00015.

- [33] Jack Turner et al. “Characterising Across-Stack Optimisations for Deep Convolutional Neural Networks”. In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 2018, pp. 101–110. DOI: 10.1109/IISWC.2018.8573503.
- [34] Bhaskar Mondal. “Artificial intelligence: state of the art”. In: *Recent trends and advances in artificial intelligence and internet of things* (2020), pp. 389–425.
- [35] Amey Thakur and Archit Konde. “Fundamentals of neural networks”. In: *International Journal for Research in Applied Science and Engineering Technology* 9.VIII (2021), pp. 407–426.
- [36] Andrej Karpathy et al. “Cs231n convolutional neural networks for visual recognition”. In: *Neural networks* 1.1 (2016).
- [37] Saturn Cloud. *A Comprehensive Guide to Convolutional Neural Networks: The ELI5 Way*. Accessed: 2024-07-24. 2023. URL: <https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/>.
- [38] S. Sumit. *A Comprehensive Guide to Convolutional Neural Networks*. <https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/>. Online; accessed 24-July-2024. Oct. 2018.
- [39] Léon Bottou. “Large-Scale Machine Learning with Stochastic Gradient Descent”. In: *Proceedings of COMPSTAT’2010*. Ed. by Yves Lechevallier and Gilbert Saporta. Heidelberg: Physica-Verlag HD, 2010, pp. 177–186. ISBN: 978-3-7908-2604-3.
- [40] Stanford University. *CS231n Convolutional Neural Networks for Visual Recognition*. Accessed: Oct. 21, 2023. 2023. URL: <https://cs231n.github.io/convolutional-networks/>.
- [41] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep Learning*. Vol. 1. Cambridge, MA, USA: MIT Press, 2017.
- [42] Günter Klambauer et al. “Self-Normalizing Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/5d44ee6f2c3f71b73125876103c8f6c4-Paper.pdf.
- [43] Dan Hendrycks and Kevin Gimpel. “Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units”. In: *CoRR* abs/1606.08415 (2016). arXiv: 1606.08415. URL: <http://arxiv.org/abs/1606.08415>.
- [44] Mark Sandler et al. “MobileNetV2: Inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Salt Lake City, UT, USA, June 2018, pp. 4510–4520.
- [45] Shibani Santurkar et al. “How Does Batch Normalization Help Optimization?” In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018, pp. 2488–2498. URL: https://proceedings.neurips.cc/paper_files/paper/2018/file/905056c1ac1dad141560467e0a99e1cf-Paper.pdf.

- [46] Ethem Alpaydin. *Introduction to machine learning*. 3rd. Cambridge, MA, USA: MIT Press, 2020.
- [47] Jahnavi Mahanta. “Introduction to neural networks, advantages and applications”. In: *Towards Data Science* 13 (2017).
- [48] Nouma Izeboudjen, C Larbes, and Ahcene Farah. “A new classification approach for neural networks hardware: from standards chips to embedded systems on chip”. In: *Artificial Intelligence Review* 41 (2014), pp. 491–534.
- [49] Ding Wang, Haibo He, and Derong Liu. “Intelligent optimal control with critic learning for a nonlinear overhead crane system”. In: *IEEE Transactions on Industrial Informatics* 14.7 (2017), pp. 2932–2940.
- [50] Haibo He and Eduardo A Garcia. “Learning from imbalanced data”. In: *IEEE Transactions on knowledge and data engineering* 21.9 (2009), pp. 1263–1284.
- [51] Ahmad Mozaffari, Mahdi Emami, and Alireza Fathi. “A comprehensive investigation into the performance, robustness, scalability and convergence of chaos-enhanced evolutionary algorithms with boundary constraints”. In: *Artificial Intelligence Review* 52 (2019), pp. 2319–2380.
- [52] Sadeque Reza Khan et al. “Commercial off-the-shelf components (COTS) in realizing miniature implantable wireless medical devices: A review”. In: *Sensors* 22.10 (2022), p. 3635.
- [53] Vivienne Sze et al. “Efficient Processing of Deep Neural Networks: A Tutorial and Survey”. In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329. DOI: 10.1109/JPROC.2017.2761740.
- [54] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. *Benchmarking TPU, GPU, and CPU Platforms for Deep Learning*. 2019. arXiv: 1907.10701 [cs.LG]. URL: <https://arxiv.org/abs/1907.10701>.
- [55] Ahmed Ghazi Blaiech et al. “A Survey and Taxonomy of FPGA-based Deep Learning Accelerators”. In: *Journal of Systems Architecture* 98 (2019), pp. 331–345. ISSN: 1383-7621. DOI: <https://doi.org/10.1016/j.sysarc.2019.01.007>. URL: <https://www.sciencedirect.com/science/article/pii/S1383762118304156>.
- [56] Kaiyuan Guo et al. “[DL] A Survey of FPGA-based Neural Network Inference Accelerators”. In: *ACM Trans. Reconfigurable Technol. Syst.* 12.1 (Mar. 2019). ISSN: 1936-7406. DOI: 10.1145/3289185. URL: <https://doi.org/10.1145/3289185>.
- [57] Theofilos Spyrou and Haralampos-G. Stratigopoulos. “On-Line Testing of Neuromorphic Hardware”. In: *2023 IEEE European Test Symposium (ETS)*. 2023, pp. 1–6. DOI: 10.1109/ETS56758.2023.10174077.
- [58] Theofilos Spyrou et al. “Reliability Analysis of a Spiking Neural Network Hardware Accelerator”. In: *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2022, pp. 370–375. DOI: 10.23919/DATE54114.2022.9774711.

- [59] Haralampos-G. Stratigopoulos, Theofilos Spyrou, and Spyridon Raptis. “Testing and Reliability of Spiking Neural Networks: A Review of the State-of-the-Art”. In: *2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 2023, pp. 1–8. DOI: 10.1109/DFT59622.2023.10313541.
- [60] T. Patrick Xiao et al. “Ionizing Radiation Effects in SONOS-Based Neuro-morphic Inference Accelerators”. In: *IEEE Transactions on Nuclear Science* 68.5 (2021), pp. 762–769. DOI: 10.1109/TNS.2021.3058548.
- [61] Q-Engineering. *Google Coral’s TPU explained*. Accessed: 2024-07-24. 2023. URL: <https://qengineering.eu/google-corals-tpu-explained.html>.
- [62] Google Cloud. *TPU Architecture*. Accessed: 2024-07-24. 2024. URL: <https://cloud.google.com/edge-tpu>.
- [63] Norman P. Jouppi et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *ISCA ’17*. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 1–12. ISBN: 9781450348928. DOI: 10.1145/3079856.3080246. URL: <https://doi.org/10.1145/3079856.3080246>.
- [64] Stylianos Venieris, Alexandros Kouris, and Christos Bouganis. “Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions”. In: *ACM Computing Surveys* 51 (Mar. 2018). DOI: 10.1145/3186332.
- [65] Yijin Guan et al. “FPGA-based accelerator for long short-term memory recurrent neural networks”. In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2017, pp. 629–634. DOI: 10.1109/ASPDAC.2017.7858394.
- [66] Ahmed Ghazi Blaiech et al. “A Survey and Taxonomy of FPGA-based Deep Learning Accelerators”. In: *Journal of Systems Architecture* 98 (2019), pp. 331–345. ISSN: 1383-7621. DOI: <https://doi.org/10.1016/j.sysarc.2019.01.007>. URL: <https://www.sciencedirect.com/science/article/pii/S1383762118304156>.
- [67] Kaiyuan Guo et al. “A survey of FPGA-based neural network inference accelerator”. In: *Journal of Signal Processing Systems* 90.5 (2018), pp. 579–598.
- [68] Shicheng Zhang et al. “TensorRT acceleration based on deep learning photoelectric target detection”. In: *International Conference on Intelligent and Human-Computer Interaction Technology (IHCIT 2022)*. Ed. by Wei Wei. Vol. 12344. International Society for Optics and Photonics. SPIE, 2022, 123 bibrangedash 4418. DOI: 10.1117/12.2655302. URL: <https://doi.org/10.1117/12.2655302>.
- [69] Zhe Jia et al. *Dissecting the Graphcore IPU Architecture via Microbenchmarking*. 2019. arXiv: 1912.03413 [cs.DC].
- [70] Mingzhen Li et al. “The Deep Learning Compiler: A Comprehensive Survey”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2021), pp. 708–727. DOI: 10.1109/TPDS.2020.3030548.

- [71] Animesh Jain et al. *Efficient Execution of Quantized Deep Learning Models: A Compiler Approach*. 2020. DOI: 10.48550/ARXIV.2006.10226. arXiv: 2006.10226 [cs.DC]. URL: <https://arxiv.org/abs/2006.10226>.
- [72] Scott Cyphers et al. “Intel ngraph: An intermediate representation, compiler, and executor for deep learning”. In: *arXiv preprint arXiv:1801.08058* (2018). arXiv: 1801.08058 [cs.DC].
- [73] Nadav Rotem et al. “Glow: Graph Lowering Compiler Techniques for Neural Networks”. In: *arXiv preprint arXiv:1805.00907* (2019). arXiv: 1805.00907 [cs.PL].
- [74] Tianqi Chen et al. “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning”. In: *arXiv preprint arXiv:1802.04799* (2018). DOI: 10.48550/ARXIV.1802.04799. arXiv: 1802.04799 [cs.LG]. URL: <https://arxiv.org/abs/1802.04799>.
- [75] Chris Leary and Todd Wang. “XLA: TensorFlow, compiled”. In: *TensorFlow Dev Summit 1.2* (2017).
- [76] Gernot Fiala, Zhenyu Ye, and Christian Steger. “Pupil Detection for Augmented and Virtual Reality based on Images with Reduced Bit Depths”. In: *2022 IEEE Sensors Applications Symposium (SAS)*. 2022, pp. 1–5. DOI: 10.1109/SAS54819.2022.9881378.
- [77] Apache TVM. *VTA Hardware - TVM Documentation*. <https://tvm.apache.org/docs/topic/vta/dev/hardware.html>. Accessed: 2024-07-24. 2024.
- [78] Zhongyi Lin, Evangelos Georganas, and John D. Owens. “Towards Flexible and Compiler-Friendly Layer Fusion for CNNs on Multicore CPUs”. In: *Euro-Par 2021: Parallel Processing*. Ed. by Leonel Sousa, Nuno Roma, and Pedro Tomás. Cham: Springer International Publishing, 2021, pp. 232–248. ISBN: 978-3-030-85665-6.
- [79] Andres Rodriguez. “Compiler Optimizations”. In: *Deep Learning Systems: Algorithms, Compilers, and Processors for Large-Scale Production*. Springer, 2021, pp. 159–176.
- [80] Li-Wen Chang et al. “Accelerating recurrent neural networks through compiler techniques and quantization”. In: *Workshop on Systems for ML and Open Source Software*. 2018.
- [81] Jie Zhao and Peng Di. “Optimizing the Memory Hierarchy by Compositing Automatic Transformations on Computations and Data”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 427–441. DOI: 10.1109/MICRO50266.2020.00044.
- [82] Xilinx. *Vitis AI - Xilinx Documentation*. Accessed: 2024-07-24. 2024. URL: <https://www.xilinx.com/products/design-tools/ai-inference/vitis-ai.html>.
- [83] Michaela Blott et al. “FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.3 (Dec. 2018). ISSN: 1936-7406. DOI: 10.1145/3242897. URL: <https://doi.org/10.1145/3242897>.

- [84] V. Loncar et al. *fastmachinelearning/hls4ml: bartsia (v0.5.0)*. Accessed: 2024-07-24. 2021. DOI: 10.5281/zenodo.4585796. URL: <https://github.com/fastmachinelearning/hls4ml>.
- [85] Farah Fahim et al. *hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices*. 2021. arXiv: 2103.05579 [cs.LG]. URL: <https://arxiv.org/abs/2103.05579>.
- [86] Thierry Moreau et al. “A Hardware–Software Blueprint for Flexible Deep Learning Specialization”. In: *IEEE Micro* 39.5 (2019), pp. 8–16. DOI: 10.1109/MM.2019.2928962.
- [87] João Silva et al. “Customizable FPGA-Based Hardware Accelerator for Standard Convolution Processes Empowered with Quantization Applied to LiDAR Data”. In: *Sensors* 22.6 (2022). ISSN: 1424-8220. DOI: 10.3390/s22062184. URL: <https://www.mdpi.com/1424-8220/22/6/21%5C-84>.
- [88] Alessandro Biondi et al. “A Safe, Secure, and Predictable Software Architecture for Deep Learning in Safety-Critical Systems”. In: *IEEE Embedded Systems Letters* 12.3 (2020), pp. 78–82. DOI: 10.1109/LES.2019.2953253.
- [89] National Transportation Safety Board. *Collision Between a Car Operating With Automated Vehicle Control Systems and a Tractor-Semitrailer Truck Near Williston, Florida, May 7, 2016*. 2017.
- [90] Christian Szegedy et al. *Intriguing properties of neural networks*. 2014. arXiv: 1312.6199 [cs.CV]. URL: <https://arxiv.org/abs/1312.6199>.
- [91] Iraj Moghaddasi, Saeid Gorgin, and Jeong-A Lee. “Dependable DNN Accelerator for Safety-Critical Systems: A Review on the Aging Perspective”. In: *IEEE Access* 11 (2023), pp. 89803–89834. DOI: 10.1109/ACCESS.2023.3300376.
- [92] A. Avizienis et al. “Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2.
- [93] Sparsh Mittal. “A survey on modeling and improving reliability of DNN algorithms and accelerators”. In: *Journal of Systems Architecture* 104 (2020), p. 101689.
- [94] Younis Ibrahim et al. “Soft errors in DNN accelerators: A comprehensive review”. In: *Microelectronics Reliability* 115 (2020), p. 113969.
- [95] Pramesh Pandey et al. “Challenges and Opportunities in Near-Threshold DNN Accelerators around Timing Errors”. In: *Journal of Low Power Electronics and Applications* 10.4 (2020). ISSN: 2079-9268. DOI: 10.3390/jlpea10-040033. URL: <https://www.mdpi.com/2079-9268/10/4/33>.
- [96] Cheng Liu et al. “Special Session: Fault-Tolerant Deep Learning: A Hierarchical Perspective”. In: *2022 IEEE 40th VLSI Test Symposium (VTS)*. 2022, pp. 1–12. DOI: 10.1109/VTS52500.2021.9794239.
- [97] Muhammad Shafique et al. “Robust Machine Learning Systems: Challenges, Current Trends, Perspectives, and the Road Ahead”. In: *IEEE Design Test* 37.2 (2020), pp. 30–57. DOI: 10.1109/MDAT.2020.2971217.

- [98] Jon Perez-Cerrolaza et al. “GPU Devices for Safety-Critical Systems: A Survey”. In: 55.7 (Dec. 2022). ISSN: 0360-0300. DOI: 10.1145/3549526. URL: <https://doi.org/10.1145/3549526>.
- [99] Yiran Chen et al. “A Survey of Accelerator Architectures for Deep Neural Networks”. In: *Engineering* 6.3 (2020), pp. 264–274. ISSN: 2095-8099. DOI: <https://doi.org/10.1016/j.eng.2020.01.007>. URL: <https://www.sciencedirect.com/science/article/pii/S2095809919306356>.
- [100] Hanan Hussain, PS Tamizharasan, and CS Rahul. “Design possibilities and challenges of DNN models: a review on the perspective of end devices”. In: *Artificial Intelligence Review* (2022), pp. 1–59.
- [101] Meriam Dhouibi et al. “Accelerating deep neural networks implementation: A survey”. In: *IET Computers & Digital Techniques* 15.2 (2021), pp. 79–96.
- [102] Riadh Ayachi, Yahia Said, and Abdessalem Ben Abdelali. “Optimizing neural networks for efficient FPGA implementation: A survey”. In: *Archives of Computational Methods in Engineering* 28.7 (2021), pp. 4537–4547.
- [103] Ayush Arunachalam et al. “Fault Resilience of DNN Accelerators for Compressed Sensor Inputs”. In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2022, pp. 329–332. DOI: 10.1109/ISVLSI54635.2022.00073.
- [104] Oludare Isaac Abiodun et al. “State-of-the-art in artificial neural network applications: A survey”. In: *Heliyon* 4.11 (2018).
- [105] Miguel Masmano, Ismael Ripoll, and Alfons Crespo. “A comparison of memory allocators for real-time applications”. In: *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems*. JTRES ’06. Paris, France: Association for Computing Machinery, 2006, pp. 68–76. ISBN: 1595935444. DOI: 10.1145/1167999.1168012. URL: <https://doi.org/10.1145/1167999.1168012>.
- [106] Xinjian Long et al. “An Intelligent Framework for Oversubscription Management in CPU-GPU Unified Memory”. In: *Journal of Grid Computing* 21.1 (Feb. 2023). Accessed: 2024-07-24, p. 11. DOI: 10.1007/s10723-023-09646-1. URL: <https://doi.org/10.1007/s10723-023-09646-1>.
- [107] Jorg Herter et al. “CAMA: A Predictable Cache-Aware Memory Allocator”. In: *2011 23rd Euromicro Conference on Real-Time Systems*. 2011, pp. 23–32. DOI: 10.1109/ECRTS.2011.11.
- [108] Nicholas Harvey-Lees-Green et al. “A Dynamic Memory Management Unit for Real Time Systems”. In: *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*. 2017, pp. 84–91. DOI: 10.1109/ISORC.2017.4.
- [109] Zhuo Tang et al. “Dynamic memory-aware scheduling in spark computing environment”. In: *Journal of Parallel and Distributed Computing* 141 (2020), pp. 10–22. ISSN: 0743-7315. DOI: <https://doi.org/10.1016/j.jpdc.2020.03.010>. URL: <https://www.sciencedirect.com/science/article/pii/S074373151930824X>.

- [110] Benny Akesson, Kees Goossens, and Markus Ringhofer. “Predator: a predictable SDRAM memory controller”. In: *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. CODES+ISSS ’07. Salzburg, Austria: Association for Computing Machinery, 2007, pp. 251–256. ISBN: 9781595938244. DOI: 10.1145/1289816.1289877. URL: <https://doi.org/10.1145/1289816.1289877>.
- [111] Scott Rixner et al. “Memory access scheduling”. In: *ACM SIGARCH Computer Architecture News* 28.2 (May 2000), pp. 128–138. ISSN: 0163-5964. DOI: 10.1145/342001.339668. URL: <https://doi.org/10.1145/342001.339668>.
- [112] Yaping Luo et al. “An architecture pattern for safety critical automated driving applications: Design and analysis”. In: *2017 Annual IEEE International Systems Conference (SysCon)*. 2017, pp. 1–7. DOI: 10.1109/SYSCON.2017.7934739.
- [113] Husheng Zhou, Soroush Bateni, and Cong Liu. “S²DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads”. In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2018), pp. 190–201. DOI: 10.1109/RTAS.2018.00028.
- [114] Ming Yang et al. “Re-Thinking CNN Frameworks for Time-Sensitive Autonomous -Driving Applications: Addressing an Industrial Challenge”. In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019, pp. 305–317. DOI: 10.1109/RTAS.2019.00033.
- [115] Yu-Hsin Chen et al. “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks”. In: *IEEE journal of solid-state circuits* 52.1 (2016), pp. 127–138.
- [116] Song Han et al. “ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA”. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 75–84. ISBN: 9781450343541. DOI: 10.1145/3020078.3021745. URL: <https://doi.org/10.1145/3020078.3021745>.
- [117] Tianshi Chen et al. “DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning”. In: *SIGARCH Comput. Archit. News* 42.1 (Feb. 2014), pp. 269–284. ISSN: 0163-5964. DOI: 10.1145/2654822.2541967. URL: <https://doi.org/10.1145/2654822.2541967>.
- [118] Yunji Chen et al. “DaDianNao: A Machine-Learning Supercomputer”. In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2014, pp. 609–622. DOI: 10.1109/MICRO.2014.58.
- [119] Ji Hua Ang et al. “Interference-less neural network training”. In: *Neurocomputing* 71.16 (2008). Advances in Neural Information Processing (ICONIP 2006) / Brazilian Symposium on Neural Networks (SBRN 2006), pp. 3509–3524. ISSN: 0925-2312. DOI: <https://doi.org/10.1016/j.neucom.2007.10.012>. URL: <https://www.sciencedirect.com/science/article/pii/S0925231207003578>.

- [120] Xuechao Wei, Yun Liang, and Jason Cong. “Overcoming Data Transfer Bottlenecks in FPGA-based DNN Accelerators via Layer Conscious Memory Management”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC ’19. Las Vegas, NV, USA: Association for Computing Machinery, 2019. ISBN: 9781450367257. DOI: 10.1145/3316781.3317875. URL: <https://doi.org/10.1145/3316781.3317875>.
- [121] Rongshan Wei et al. “Memory Access Optimization of a Neural Network Accelerator Based on Memory Controller”. In: *Electronics* 10.4 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10040438. URL: <https://www.mdpi.com/2079-9292/10/4/438>.
- [122] Rodolfo Pellizzoni et al. “Worst case delay analysis for memory interference in multicore systems”. In: *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*. 2010, pp. 741–746. DOI: 10.1109/DATE.2010.5456952.
- [123] Heechul Yun et al. “Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality”. In: *2012 24th Euromicro Conference on Real-Time Systems*. 2012, pp. 299–308. DOI: 10.1109/ECRTS.2012.32.
- [124] Chang Joo Lee et al. “Prefetch-Aware Memory Controllers”. In: *IEEE Transactions on Computers* 60.10 (2011), pp. 1406–1430. DOI: 10.1109/TC.2010.214.
- [125] Xiaofei Liao et al. “Improving Bank-Level Parallelism for In-Memory Checkpointing in Hybrid Memory Systems”. In: *IEEE Transactions on Big Data* 8.2 (2022), pp. 289–301. DOI: 10.1109/TBDATA.2018.2865964.
- [126] Roman Obermaisser et al. *Time-triggered communication*. Vol. 19. CRC Press Boca Raton, FL, USA, 2012.
- [127] Florian Franzmann et al. “From Intent to Effect: Tool-Based Generation of Time-Triggered Real-Time Systems on Multi-core Processors”. In: *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*. 2016, pp. 134–141. DOI: 10.1109/ISORC.2016.27.
- [128] Philipp Ittershagen, Kim Grüttner, and Wolfgang Nebel. “An Integration Flow for Mixed-Critical Embedded Systems on a Flexible Time-Triggered Platform”. In: *ACM Trans. Des. Autom. Electron. Syst.* 23.4 (May 2018). ISSN: 1084-4309. DOI: 10.1145/3190837. URL: <https://doi.org/10.1145/3190837>.
- [129] Yosab Bebawy, Aniebiet Micheal Ezekiel, and Roman Obermaisser. “Superscalar Time-Triggered Versatile-Tensor Accelerator”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 44.7 (2025), pp. 2503–2515. DOI: 10.1109/TCAD.2025.3528355.
- [130] Mohamed Hassan, Anirudh M. Kaushik, and Hiren Patel. “Predictable Cache Coherence for Multi-core Real-Time Systems”. In: *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2017, pp. 235–246. DOI: 10.1109/RTAS.2017.13.
- [131] Edward A. Lee. “The problem with threads”. In: *Computer* 39.5 (2006), pp. 33–42. DOI: 10.1109/MC.2006.180.

- [132] Martin Maas et al. “TelaMalloc: Efficient On-Chip Memory Allocation for Production Machine Learning Accelerators”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2022, pp. 123–137. ISBN: 9781450399159. DOI: 10.1145/3567955.3567961. URL: <https://doi.org/10.1145/3567955.3567961>.
- [133] Yishen Chen et al. “BHive: A Benchmark Suite and Measurement Framework for Validating x86-64 Basic Block Performance Models”. In: *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 2019, pp. 167–177. DOI: 10.1109/IISWC47752.2019.9042166.
- [134] Nicholas C. Doyle et al. “Performance impacts and limitations of hardware memory access trace collection”. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 2017, pp. 506–511. DOI: 10.23919/DATE.2017.7927041.
- [135] Wilfried Elmenreich, Günther Bauer, and Hermann Kopetz. *The time-triggered paradigm*. Citeseer, 2003.
- [136] Terrence J Sejnowski. “Memory and neural networks”. In: *Memory* (1998), pp. 162–181.
- [137] Chao Li et al. “Optimizing memory efficiency for deep convolutional neural networks on GPUs”. In: *SC’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2016, pp. 633–644.
- [138] Aniebiet Micheal Ezekiel, Daniel Onwuchekwa, and Roman Obermaisser. “Optimization of the Versatile Tensor Accelerator (VTA) Load Module in a Time-Triggered Memory Access”. In: *2023 26th Euromicro Conference on Digital System Design (DSD)*. 2023, pp. 146–152. DOI: 10.1109/DSD60849.2023.00030.
- [139] Xiaowen Chen et al. “Round-trip DRAM Access Fairness in 3D NoC-based Many-core Systems”. In: *ACM Trans. Embed. Comput. Syst.* 16.5s (Sept. 2017). ISSN: 1539-9087. DOI: 10.1145/3126561. URL: <https://doi.org/10.1145/3126561>.
- [140] Ayman Gendy, Lei Dong, and Michael Pont. “Improving the Performance of Time-Triggered Embedded Systems by Means of a Scheduler Agent”. In: (Jan. 2007). DOI: 10.1115/DETC2007-35369.
- [141] Rakotojaona Nambinina, Daniel Onwuchekwa, and Roman Obermaisser. “Latency-Aware Frequency Scaling in Time-Triggered Network-on-Chip Architecture”. In: *2023 7th International Conference on Computing Methodologies and Communication (ICCMC)*. 2023, pp. 1480–1488. DOI: 10.1109/ICCMC56507.2023.10084313.
- [142] Igor Freire et al. “Clock Synchronization Algorithms Over PTP-Unaware Networks: Reproducible Comparison Using an FPGA Testbed”. In: *IEEE Access* 9 (2021), pp. 20575–20601. DOI: 10.1109/ACCESS.2021.3054164.

- [143] Mohamed Amine Khelassi and Yasmina Abdeddaïm. “Leveraging mixed criticality task budgets”. In: *International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. Padova, Italy, Sept. 2024. URL: <https://hal.science/hal-04667081>.
- [144] Jason Cong et al. *Best-Effort FPGA Programming: A Few Steps Can Go a Long Way*. 2018. arXiv: 1807.01340 [cs.AR]. URL: <https://arxiv.org/abs/1807.01340>.
- [145] H.C. Kim et al. “At-speed interconnect test and diagnosis of external memories on a system”. In: *2004 International Conference on Test*. 2004, pp. 156–162. DOI: 10.1109/TEST.2004.1386948.
- [146] Amir Gholami et al. “A survey of quantization methods for efficient neural network inference”. In: *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, pp. 291–326.
- [147] Zhenyuan Ruan et al. “ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2018, pp. 9–16. DOI: 10.1109/FCCM.2018.00011.
- [148] Peter Braun and Heiner Litz. “Understanding Memory Access Patterns for Prefetching”. In: *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA* (Apr. 2019). URL: <https://par.nsf.gov/biblio/10187649>.
- [149] Byunghyun Jang et al. “Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures”. In: *IEEE Transactions on Parallel and Distributed Systems* 22.1 (2011), pp. 105–118. DOI: 10.1109/TPDS.2010.107.
- [150] Yosab Bebawy et al. “Time-Triggered Inference on FPGAs”. In: *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2024, pp. 1–7. DOI: 10.1109/ETFA61755.2024.10710675.
- [151] Yash Akhauri et al. “RHNAS: Realizable Hardware and Neural Architecture Search”. In: *ArXiv* abs/2106.09180 (2021).
- [152] Angshuman Parashar et al. “SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks”. In: *SIGARCH Comput. Archit. News*. ISCA ’17 45.2 (June 2017), pp. 27–40. ISSN: 0163-5964. DOI: 10.1145/3140659.3080254. URL: <https://doi.org/10.1145/3140659.3080254>.
- [153] Yijin Guan et al. “FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates”. In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2017, pp. 152–159. DOI: 10.1109/FCCM.2017.25.
- [154] Chen Zhang et al. “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks”. In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’15. Monterey, California, USA: Association for Computing Machinery, 2015, pp. 161–170. ISBN: 9781450333153. DOI: 10.1145/2684746.2689060. URL: <https://doi.org/10.1145/2684746.2689060>.

- [155] Min Zhang et al. “Optimized Compression for Implementing Convolutional Neural Networks on FPGA”. In: *Electronics* 8.3 (2019). ISSN: 2079-9292. DOI: 10.3390/electronics8030295. URL: <https://www.mdpi.com/2079-9292/8/3/295>.
- [156] Xiaoli Zhang et al. “An Efficient FPGA-Based Implementation for Quantized Remote Sensing Image Scene Classification Network”. In: *Electronics* 9.9 (2020). ISSN: 2079-9292. DOI: 10.3390/electronics9091344. URL: <https://www.mdpi.com/2079-9292/9/9/1344>.
- [157] Chaoyang Zhu et al. “An Efficient Hardware Accelerator for Structured Sparse Convolutional Neural Networks on FPGAs”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.9 (2020), pp. 1953–1965. DOI: 10.1109/TVLSI.2020.3002779.
- [158] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: <https://arxiv.org/abs/1512.03385>.
- [159] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. “DRAMSim2: A Cycle Accurate Memory System Simulator”. In: *IEEE Computer Architecture Letters* 10.1 (2011), pp. 16–19. DOI: 10.1109/L-CA.2011.4.
- [160] Konstantin Bick et al. “Fast and Accurate Memory Simulation by Integrating DRAMSim2 into McSimA+”. In: *Electronics* 7.8 (2018). ISSN: 2079-9292. URL: <https://www.mdpi.com/2079-9292/7/8/152>.
- [161] Ayaz Akram and Lina Sawalha. “A comparison of x86 computer architecture simulators”. In: (2016).

List of Acronyms

AI	Artificial Intelligence
ASIC	Application-Specific Integrated Circuit
CA	Certification Authority
CAIS	Critical Autonomous AI-based Systems
CNNs	Convolutional Neural Networks
COTS	Commercial-Off-The-Shelf
CPU	Central Processing Unit
DAL	Design-Assurance Level
DDR	Dual-Data Rate
DRAM	Dynamic Random Access Memory
EASA	European Aviation Safety Agency
ET	Event-Triggered
FAA	Federal Aviation Administration
FINN	FINN
FPGA	Field-Programmable Gate Array
FPGAs	Field-Programmable Gate Arrays
FR-FCFS	First-Ready First-Come-First-Serve
GPCS	General-Purpose Computing System
HRT	Hard Real-Time
I/O	Input/Output
MA	Memory Access
MAT	Memory Access Time
NNs	Neural Networks
PYNQ	Python Productivity for Zynq.

RT Real-Time

SC Safety-Critical

SCS Safety-Critical System

SRAM Static Random Access Memory

SSD Solid-State Drive

TT Time-Triggered

TTmaVTA Time-Triggered Memory Access Versatile Tensor Accelerator (VTA)

OPTTmaVTA Versatile Tensor Accelerator(VTA) with Time-Triggered Memory Access Optimisations

TVM Tensor Virtual Machine

VTA Versatile Tensor Accelerator

WCET Worst-Case Execution Time

GPUs Graphics Processing Units

ASICs Application-Specific Integrated Circuits

TPUs Tensor Processing Units

DPUs Data Processing Units

HPC high-performance computing

FINN Fast, Scalable Quantized Neural Network Inference

VITISAI Vitis Artificial Intelligence

HLS4ML High-Level Synthesis for Machine Learning

CUDA Compute Unified Device Architecture

ROCm Radeon Open Compute

SMs streaming multiprocessors

Appendix

A State-Machine of Time-Triggered Memory Access VTA (TTmaVTA)

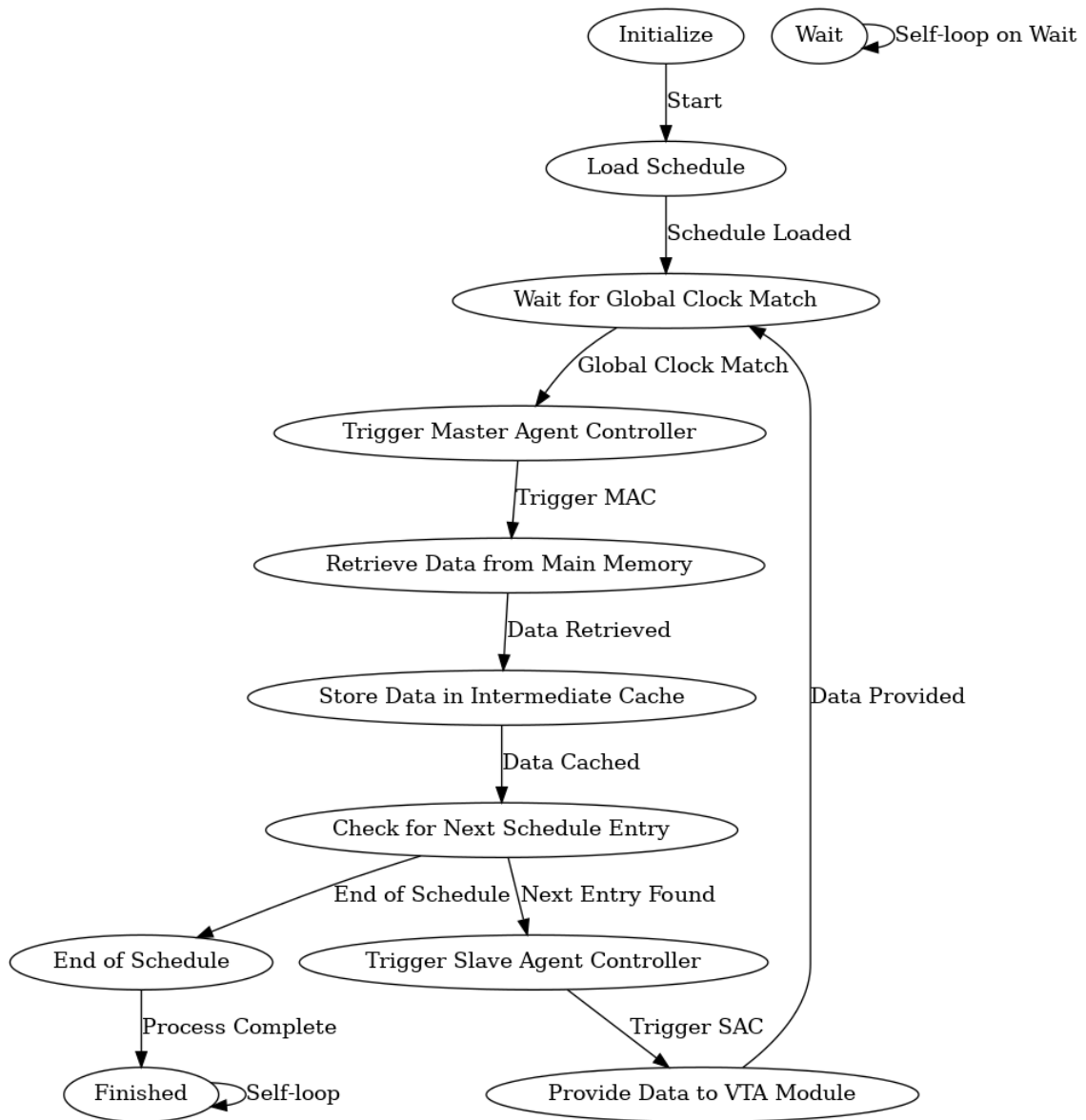


Figure 15: TTmaVTA Task Coding State-machine

B Caching Algorithm for Optimisation

Algorithm 10 Intermediate Caching Optimisation for Write-Read Pairs

```

1: procedure OPTIMISECACHE(Memory Access Trace (Address, OperationType, DataSize,
   StartTime, FinishTime))
2:   Initialise CacheID  $\leftarrow$  0
3:   Initialise CacheTable  $\leftarrow$  {}
4:   for each operation in Memory Access Trace do
5:     if operation.OperationType == "Write" then
6:       WriteAddress  $\leftarrow$  operation.Address
7:       DataSize  $\leftarrow$  operation.DataSize
8:       WriteTime  $\leftarrow$  operation.StartTime
9:       Check for next Read Memory Operation with the same Address
10:      if Write-Read pair exists then
11:        Check status of cached data in primary cache
12:        if cached data is read then
13:          if current Write-Read pair has different Address then
14:            Flush Cache and write new cache data
15:          else
16:            Cache Write data in Secondary Cache
17:          end if
18:        end if
19:        CacheTable[WriteAddress]  $\leftarrow$  {WriteData, WriteTime, CacheID}
20:        CacheID  $\leftarrow$  CacheID + 1
21:      else
22:        Proceed with normal Write operation and update CacheTable
23:      end if
24:    else if operation.OperationType == "Read" then
25:      if WriteAddress  $\in$  CacheTable then
26:        if ReadFrequency  $\geq$  25% before next Write then
27:          Retrieve CacheID from CacheTable[WriteAddress]
28:          Redirect Read operation to CacheID
29:          Set EntryBit  $\leftarrow$  1  $\triangleright$  Mark as cacheable
30:        end if
31:        Fetch read data from Primary cache
32:        NextStartTime = LastFinishTime - current duration
33:      else
34:        if memory bus is free then
35:          NextStartTime = LastFinishTime
36:          Start next Memory Operation
37:        end if
38:      end if
39:      if a new Write operation occurs at WriteAddress then
40:        CacheTable[WriteAddress]  $\leftarrow$  None  $\triangleright$  Invalidate previous cache data
41:        CacheTable[WriteAddress]  $\leftarrow$  {NewWriteData, NewWriteTime, CacheID}
42:      end if
43:    end if
44:  end for
45:  for each operation in Memory Access Trace do
46:    if operation is cacheable then
47:      Set EntryBit  $\leftarrow$  1
48:    else
49:      Set EntryBit  $\leftarrow$  0
50:    end if
51:  end for
52:  Append EntryBit to the updated schedule
53:  Simulate the updated schedule and Compute time performance improvements
54: end procedure

```

C Prefetching Algorithm for Optimisation

Algorithm 11 Prefetching Data During Memory Idle Periods

```

1: procedure PREFETCHINGDATA
2:   if predecessor operations end and next MemOps have dependencies then
3:     while not all dependencies are met do
4:       Wait until the dependency is resolved
5:        $LastFinishTime \leftarrow$  FinishTime of the dependency
6:       if memory bus is free then
7:          $NextStartTime \leftarrow LastFinishTime$ 
8:         Start the next Memory Operation
9:       end if
10:    end while
11:  else
12:    if memory bus is free then
13:       $NextStartTime \leftarrow LastFinishTime$ 
14:      Start the next Memory Operation
15:    end if
16:  end if
17:  if memory idle instance then
18:    if memory bus is free then
19:      if next Memory Operation has unfulfilled dependencies then
20:        if second-next MemOps has unfulfilled dependencies then
21:          if second-next MemOps has fulfilled predecessors then
22:            while not all dependencies are met do
23:              Wait until the dependency is resolved
24:               $LastFinishTime \leftarrow$  FinishTime of the dependency
25:            end while
26:          else
27:            Prefetch the second-next MemOp ahead of request
28:            Store in prefetched cache
29:             $NextStartTime \leftarrow LastFinishTime - \text{current duration}$ 
30:          end if
31:        else
32:          while not all dependencies are met do
33:            Wait until the dependency is resolved
34:             $LastFinishTime \leftarrow$  FinishTime of the dependency
35:          end while
36:        end if
37:         $NextStartTime \leftarrow$  MemoryIdlePeriod
38:        Start the next Memory Operation
39:      end if
40:    end if
41:  end if
42: end procedure

```
