8-2022

A Memory-Centric Customizable Domain-Specific FPGA Overlay for Accelerating Machine Learning Applications

Atiyehsadat Panahi
University of Arkansas, Fayetteville

A Memory-Centric Customizable Domain-Specific FPGA Overlay for Accelerating Machine Learning Applications

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Engineering, with a concentration in Computer Engineering

by

Atiyehsadat Panahi
Sharif University of Technology
Bachelor of Science in Computer Engineering, 2014
Shahid Beheshti University
Master of Science in Computer Engineering, 2016

July 2022
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council.

David Andrews, Ph.D,
Dissertation Director

Mark E. Arnold, Ph.D,
Committee Member

John M. Gauch, Ph.D,
Committee Member

Miaoqing Huang, Ph.D,
Committee Member

Abstract

Low latency inferencing is of paramount importance to a wide range of real time and user facing Machine Learning (ML) applications. Field Programmable Gate Arrays (FPGAs) offer unique advantages in delivering low latency as well as energy efficient accelertors for low latency inferencing. Unfortunately, creating machine learning accelerators in FPGAs is not easy, requiring the use of vendor specific CAD tools and low level digital and hardware microarchitecture design knowledge that the majority of ML researchers do not possess. The continued refinement of High Level Synthesis (HLS) tools can reduce but not eliminate the need for hardware-specific design knowledge. The designs by these tools can also produce inefficient use of FPGA resources that ultimately limit the performance of the neural network. This research investigated a new FPGA-based software-hardware codesigned overlay architecture that opens the advantages of FPGAs to the broader ML user community. As an overlay, the proposed design allows rapid coding and deployment of different ML network configurations and different data-widths, eliminating the prior barrier of needing to resynthesize each design. This brings important attributes of code portability over different FPGA families. The proposed overlay design is a Single-Instruction-Multiple-Data (SIMD) Processor-In-Memory (PIM) architecture developed as a programmable overlay for FPGAs. In contrast to point designs, it can be programmed to implement different types of machine learning algorithms. The overlay architecture integrates bit-serial Arithmetic Logic Units (ALUs) with distributed Block RAMs (BRAMs). The PIM design increases the size of arithmetic operations and on-chip storage capacity. User-visible inference latencies are reduced by exploiting concurrent accesses to network parameters (weights and biases) and partial results stored throughout the distributed BRAMs. Run-time performance comparisons show that the proposed design achieves a speedup compared to HLS-based or custom-tuned equivalent designs. Notably, the proposed design is programmable, allowing rapid design space exploration without the need to resynthesize when changing ML algorithms on the FPGA.

Acknowledgements

I am deeply grateful to my supervisor Dr. David Andrews for his assistance, inspiration, and helpful guidelines at every stage of the research project. He has been so supportive and understanding to me during these years. I also thank my committee members, Dr. Arnold, Dr. John Gauch, and Dr. Huang. I appreciate the time they spent on reviewing and evaluating my work. Thank you to the National Science Foundation (NSF) for the financial supports they provided under grant number 1956071 to the Computer Systems Design laboratory. I also want to thank my lab mates, Suhail Balsalama and Ange-Thierry Ishimwe for working with me during the first stages of this project and their contributions on it. Finally, I would like to thank my husband and my son for their love and continues support in the past and the future. I also thank my family and family-in-law for their patience and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine Learning (ML) algorithms have permeated every aspect of our daily lives, from how our search engines provide us with relevant and customized information, companies tailor individualized marketing campaigns, doctors access our health, through the realization of autonomous vehicles. In somewhat of a twist of fate, the rise of Machine Learning occurred as our ability to ride Moore's law began to slow down. Just as computationally intensive machine learning algorithms began positioning themselves as the next generation computationally challenging workload our ability to deliver transparent performance increases began to vanish. These two realities served as a catalyst for computer architects to explore new technologies and architectures capable of delivering the level of scalable performance necessary to meet current and next generation Machine Learning application requirements.

Field Programmable Gate Arrays (FPGAs) as well as Application-Specific Integrated Circuits (ASICs) have found their way into data centers and cloud infrastructures to meet the performance and energy efficiency requirements of current and next generation Machine Learning applications. FPGAs cannot compete with custom ASICs in terms of clock frequency and performance. However, they do combine a compelling energy efficiency argument with a unique ability to allow their gates to be reconfigured on a per-application basis after deployment [1]. Once deployed gates can be reconfigured in the field to support follow on changes to base compute algorithms or the integration of additional user functionality which can occur weekly in data centers. Such flexibility and ability to rapidly update is not a hallmark of ASICs that once deployed cannot be changed, and modifications requiring $10's of Millions of dollars of refabrication costs spanning multiyear design cycles. This delay could render the new chip obsolete and multiple generational changes behind by the time it was deployed.

As an example, Microsoft developed Catapult, an FPGA-based system to accelerate their Bing web search service [2]. Microsoft could have created an ASIC but chose to use FPGAs for

their exact ability to support the rapid updates needed to continually provide competitive advantages over other search engines. Microsoft followed up the Catapult work with Brainwave, which uses FPGAs within data centers to accelerate ML algorithms. Microsoft has deployed over 1.5 Million FPGAs throughout their data centers [7].

Forecasts predict the number of worldwide IoT edge devices will approach 75 billion in 2025, representing a fivefold increase in ten years [6]. This explosive growth is changing how we will store and analyze data. Traditional cloud-based IoT paradigms can lead to prohibitively long latencies for gathering and transferring raw data from distributed sensors to consolidated data centers for analysis and transferring actionable knowledge back to edge devices that need to take actions in real-time. Processing is moving out to where the data is produced, domain-specific hardware accelerators are becoming ubiquitous infrastructure, and latency is replacing throughput as the driving system performance requirement. FPGAs are poised to play a key role in this migration.

Despite their appeal of energy efficiency and in field customization, the poor levels of designer productivity offered by vendor development tools have left software developers and programmers reluctant to embrace their use. The success of our software industry was built on the three tenets of software engineering; abstraction, portability and reuse. FPGA hardware design violates all three of these tenets. Designers must code to the physical architecture not a higher level abstract machine model. The designs produced are machine specific and not portable, and the code must be changed for different logic families.

1.1 Research Goals and Approach

The goal of this research was to explore if the same advantages brought to software development through abstraction, portability and reuse could be brought over the design of machine learning accelerators within FPGAs.

Operating Systems enable code portability and reuse by separating high level user accessible policies from lower level platform specific mechanisms. Overlays were originally developed for

FPGAs to provide the same type of separation between policies and mechanisms. Whereas Operating Systems provide system services, Overlays break up the direct translation of application code into low level platform specific gates. In essence an overlay is an abstract model that like an operating system provides the transition between user accessible policies that are portable and can be reused over different lower level platform specific mechanisms.

Prior research on Overlays have shown that portability and reuse of the user code could be achieved by at a rather significant cost of performance. This limited their use in the performance oriented world of reconfigurable computing. The first goal of this research was to develop a new overlay architecture that could bring the best of both worlds; designer productivity levels associated with software development but levels of performance associated with custom hardware design. The research conducted for this dissertation resulted in a new memory-centric (Processor-In-Memory (PIM)) overlay architecture. In a memory-centric architecture, on-chip memories and not the Processing Elements (PEs) became the focal core of the architecture. This inverted the classic Von Neumann model, which had the processor as the focal core and led to the classic Von Neumann bottleneck. Results provided in this dissertation show that this paradigm shift in models was not just competitive with custom designs but in some cases resulted in new levels of reduced inference latency. An interesting outcome of this research is showing how the new PIM architecture can eliminate the classic Von Neumann bottleneck and enable FPGA IoT devices to meet stringent real time inference latency requirements [8].

How to configure processing elements around the on-chip memories (BRAMs) to increase storage capacity and maximize concurrent accesses become a crucial part of my investigations. Further, I explored how the memory-centric approach improved performance using memories that are optimized for data locality. The memory-centric method maximized utilization of limited FPGA resources without any complex workload scheduler. Figure 1(b) illustrates how each groups of PEs have concurrent access to local memory (BRAMs), and how data moves by hopping through the PEs core-to-core. This architecture lowered inter-PE communication latencies by allowing memory-to-memory direct transfers between the PEs' register files. This

distributed memory architecture maximizes energy efficiency as it can take high parallelism when compared with the centered memory architecture shown in Figure 1(a). PIM processors are tightly coupled with the memory, which supports the unique property of scaling the processing capability with the amount of available memory. In addition, since the processors are tightly coupled with the memory, the data does not have to move across a shared bus. In the this method, there is no buffer in the processors and the intermediate results are stored in memory. Therefore, the processors are as small as possible, and as the size of the storage increases, processing capabilities also increase to guarantee linear scaling with the memory size.

Prior investigations of overlays for Machine learning were constrained to only support a single type of neural network configuration such as a Multi-Layer Perceptron (MLP), a Convolutional Neural Network (CNN), a Long-Short Term Memory (LSTM), or a Gated Recurrent Unit (GRU) network. Prior to this research there were no studies of Overlays that could be used to support mutliple types of networks. A goal of this research was to explore the feasibility of a single generalized overlay that could support all types of network configurations. To promote code portability and reuse the new overlay architecture would need to be in a form implementable in the user logic of various FPGA families. To achieve this a parameterized overlay was developed capable of supporting various fixed-point formats and precisions. Designers can specify the fixed-point precision based on their accuracy requirements using a single overlay. Regardless of the chosen specific platform implementation mechanisms user code was not affected. This provided the requisite separation of policy from mechanism that had been missing in prior FPGA desgin flows.

As part of this research, a new run-time application programming interface (API) was developed that allowed any neural network configuration to be compiled on the overlay. Each time a new ML network configuration or a new set of network parameters is given, instead of having to resynthesize a new accelerator, the new network can be expressed in a traditional high-level programming language, compiled and linked with a set of macro libraries to retarget the overlay. This extends the prior work on FPGA overlays for machine learning algorithms that accelerated

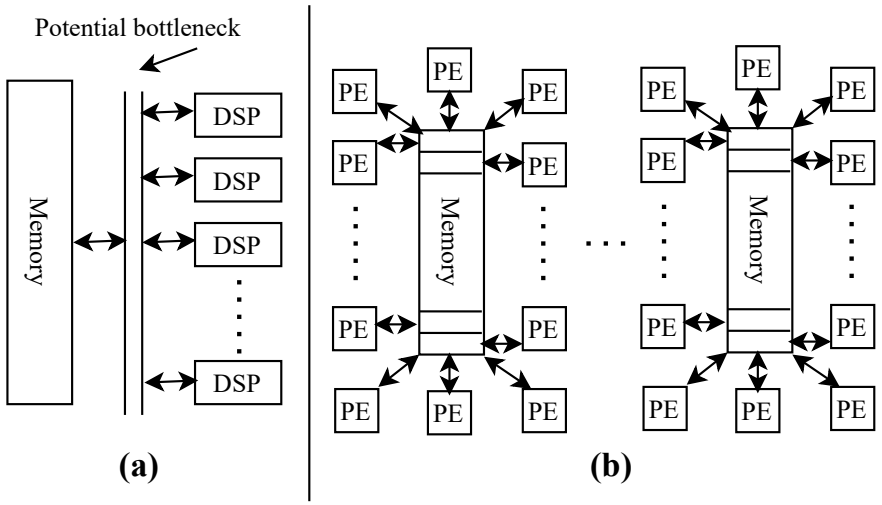

Figure 1: Memory Architecture (a) Central memory vs. (b) Our memory-centric.

only one particular network type [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]. ML algorithms continue to rapidly evolve, and various model structures and optimization techniques are constantly emerging. Perhaps the most significant outcome of this research is a new easy-to-use software programmable framework that can be used by algorithmists and software engineers as they seek to explore how new machine learning algorithms can be accelerated.

Taken together, my approach will bring software levels of productivity to the design of FPGA-based neural networks and open the use of FPGAs to the large cadre of programmers with no hardware design expertise.

1.2 Thesis Statement

When I set out on this research I proposed that a memory-centric or PIM architecture could reduce inference latencies better than what current Von Neumann type architectures with separate processing and memory subsystems could achieve. Further I put forth the idea that a PIM architecture could also serve as a generalized programmable overlay that can support all types of neural networks with performance that would remain competitive with custom designs. I proposed that supporting both communication-bound and computation-bound ML algorithms [24] on a generalized programmable overlay would require new optimizations to fit the overlay

into FPGA devices with limited available resources and on-chip memories.

The results of this research verify that a PIM architecture approach does reduce the inference latency when compared to classical Von Neumann custom accelerators. This result held on when comparing the generalized programmable overlay with other state of the art HLS generated Von Neumann architectures. The PIM architeture allows exploitation of much higher degrees of on-chip concurrent data accesses between the local memory and the processing elements compared to the HLS generated Von Neumann architectures. Results showed that concurrent data accesses scaled with the volume of data stored within the limited on-chip memories. New approaches were developed to increase the utilization of the on-chip distributed memories to achieve this scaling. Increasing the volume of data stored on-chip had the secondary affect of reducing the overhead associated with batch transferring the ML network parameters between external DRAM and on-chip BRAM buffers. Thus, a key component to reducing end-to-end inference latency was the creation of a PIM architecture that maximized concurrency of operand accesses and minimized the number of cycles spent stalled in the arithmetic units waiting for data to be transferred between the internal and external memories. New design techniques were developed as part of this research that minimized data access times by increasing the usable capacity of the on-chip BRAMs. This allowed a greater number of ML network parameters to be stored on-chip and concurrently accessed by large numbers of processing elements.

1.3 Evaluation Strategy

The claims of portability and re-programmability of the overlay were validated by running a wide suite of benchmarks gathered from the literature on the overlay. The benchmarks were either compiled or hand coded in MicroBlaze assembly. A small set of functions that represent Domain Specific Language was created during this research to facilitate rapid testing of the overlay over different FPGA architectures. All benchmarks were run on the overlay without having to re-synthesize the design. Benchmarks from the three different types of ML networks (MLP, CNN, LSTM, and GRU) benchmarks from the literature were adopted to provide as fair comparisons of

performance and resource utilization. These benchmarks were chosen to evaluate the design's performance for ML algorithms that ran the spectrum of being communication-bound or computation-bound. The benchmarks contained a suite of networks with varying numbers of network parameters ranging from sufficiently small to enable all parameters to fit into the on-chip memory, up to benchmarks with large numbers of parameters that could not fit into on-chip memory. These larger networks allowed the effects of our memory-centric method on data access times between on chip BRAM and off chip DRAM to be evaluated. Results are also included that show how data bit-widths effect the size of the overlay that could be embedded within different FPGA logic families and the resulting end to end inference latency. Detail are provided in Chapter 4.

1.4 Summary

- Developed an SoC architecture that includes a fully programmable memory-centric FPGA overlay. The overlay can implement any standard ML architecture in any data-width on any FPGA. The overlay resolve prior issues of poor design reuse and low productivity without a loss of performance.

- Defined a hierarchical design strategy that allows the overlay to be tuned to exploit the specific resources of different FPGA devices. The memory-centric architecture reduces both computation and communication latencies with the 2-D array of mixed variable and low-precision multiply-accumulate units.

- Reduced communication latencies using memory-centric computing through combining local data storage and massively parallel processing elements by enabling full concurrent accesses for ALUs to weights and partial results stored in on-chip BRAMs.

- Developed new optimizations that can be used to reduce communication latencies for matrix reduction operations while optimizing ALU performance by supporting bit-slicing in addition to bit-serial operations. These optimizations reduce the overall latency of the

ML benchmarks and increases the utilized resource utilization.

- Implemented an MLP back-propagation algorithm to support online training of MLP networks.
- Performed design space exploration on different design parameters using a set of equations than can find the optimum design for a specific ML application.
- Analysed performance and results of standard ML benchmarks including MLP, CNN, LSTM, and GRU implemented on Xilinx Virtex-7 and Virtex Ultra FPGAs. The results show that the overlay's latency is competitive and for some benchmarks lower than the previous custom ML accelerators.

The remainder of this dissertation is organized as follows. Chapter two provides a survey of related works. The description of our approach with the design's implementation details is provided in Chapter three. The next chapter discusses the results of comparing our design with previous works as well as the evaluation of the design in different metrics including the area, latency, and performance. The fifth chapter describes the optimizations have been applied on the original design and, finally, the last Chapter is a summary of the dissertation.

Chapter 2

Related Works

The FPGA overlays are developed to support software programmability for the hardware designs. Some overlays are implemented in high-level languages to generate HDL codes, and some are written in HDL languages. In the HDL-generator overlays [16, 19, 23, 45], the design is translated into RTL/HDL either manually, using automated tools, or by generating the hardware implementations with RTL-HLS hybrid templates. The other type of overlays are reprogrammed at run-time to rapidly implement different network topologies. These overlays, including our design, are usually hardware/software co-designed using an automation flow to directly compile high-level network definitions (such as Caffe or Tensorflow) to the final FPGA accelerator. In these designs [14, 15, 17, 18, 20], the hardware design synthesis is conducted once, for all network topologies. The soft processor, however, runs separately for each ML network topology. Among these works, [18, 45] support various data-widths that is also supported in our design. Our processor array overlay indicates that overlays are viable across ML applications and can compete in performance with custom designs. For a better comparison with previous works, a summary of FPGA-based overlays is presented in Table 1. As presented in this table, while our FPGA-based accelerators in the literature mainly focused on CNNs as they are computation-bound, there is significant intermediate data communication overhead in other machine learning models, diminishing overall performance gains. Therefore, the models that are communication-bound can become the new bottleneck [48]. In other words, most prior works have focused on increasing computational density, whereas in our method, the focus is on addressing this issue to well support communication bound ML algorithms using the memory-centric approach. The previous works reduce the off-chip communication by compression, sparsity, or designing with lower bit-widths [49, 50, 51, 52, 53]. However, in our processor array overlay, to maintain a deterministic execution time, we do not perform such optimizations. Instead, since we utilize a locality-aware processing-in-memory technique, we

reduce the off-chip data access time by avoiding frequently accessing off-chip memory.

In Table 1, the [22] study proposes an FPGA overlay for ML applications on the edge devices. The authors have optimized their design around the DSP blocks. They have also optimized the overlay interconnects for the patterns required by the application domain to reduce the resource utilization. The overlay presented in [14] is an HLS-based design for accelerating MLPs. Their overlay can be reprogrammed at run time to rapidly change network topologies using a linear array of processing elements. The [15] proposes an overlay for MLP fully-connected networks that is multiplication-free. Their design's inputs and activation functions are quantized to power-of-two values that enables using logical shift operations instead of multiplications. The [16] proposes a flexible debug overlay family for ML applications. Their overlay is added to the design at compile time and is configured at debug time to keep track of network parameters. The configuration can be changed between debug iterations [16]. The [17] implements a CNN accelerator using a hardware/software co-designed library. They also design an automated flow that directly compiles network high-level definitions to the CNN accelerator. In the [18], they implement an FPGA overlay using a sophisticated domain-specific graph compiler. In their method, the ML application is compiled to target the hardware overlay. In [19], they propose an overlay to bridge the gap between software and hardware development. Their design is optimized using a fine-grained layer-based pipeline architecture and a column-based cache method to improve resource utilization and latency. The [20] proposes a software-hardware co-designed for light-weight CNNs using reformulating and decomposing the operations for an efficient acceleration. In [23], a CNN accelerator is developed that automatically compiles the CNN algorithms into executable codes, which are loaded and executed by the overlay without FPGA reconfiguration. In their design, the granularity of instruction is optimized for performance and flexibility. The [45] study proposes an end-to-end framework that generates the hardware design of an MLP network form its software-based TensorFlow implementation using RTL-HLS hybrid templates. Another fine-grained FPGA overlay is designed in [64] that can be implemented in different FPGA families from different vendors. This design is optimized by

mapping the interconnection network directly into the switch fabric of the hosting FPGA. The [65] work investigates the use of a programmable overlay to reduce reconfiguration time and increase the performance of variable DSP workloads being executed on FPGAs. An FPGA overlay is presented in [66] that supports polynomial run-time mapping of dataflow applications in high-performance CPU-FPGA platforms. It also maps multi-threading onto an overlay architecture and provides the infrastructure to explore different accelerator designs [66]. The feasibility of using a coarse-grain overlay for FPGA-based acceleration of the soft processor codes is investigated in [13]. They use virtual dynamically reconfigurable method to rapidly configure the soft processor at run time for implementing a given data flow graph. In [67], the authors present an FPGA overlay architecture that is an open-source cross-compatible architecture. The [21] study proposes a highly-scalable overlay that is optimized based on the structure of FPGAs to achieve high operating frequencies. What distingushes our design from the previous designs is that our overlay supports all types of ML architectures, all data-widths, and can run on all FPGA families. Moreover, compared to the previous FPGA overlays, our memory-centric design results in efficient on-chip memory utilization and supports large ML benchmarks.

In our design, we also explore utilizing bit-serial operations based on SIMD memory-centric processor-in-memory (PIM) tiles. Early works on PIM designs developed SIMD processor arrays for video applications [54, 55, 56, 57, 58, 25]. The communication and computational requirements of machine learning have renewed interest in PIM computing architectures for ML applications [24]. Similar to our method, the [25] work proposes enhancing the ubiquitous FPGA BRAMs with in-memory compute-capabilities based on bit-serial arithmetic. The authors propose a reconfigurable in-memory accelerator architecture for deep learning inference acceleration that outperforms the Microsoft's BW [42]. The advantage of our design is supporting all types of ML networks, where as the implemented design in [25] only supports RNN networks. SIMD processor arrays are a natural approach for implementing data parallelism within the ML applications [59, 60, 61, 62, 63]. Our design extends these ideas by tightly coupling processor

ALUs with memory to address off-chip communications latencies for communication-bound ML algorithms.

Performance and area trade-off for bit-serial compared to bit-parallel arithmetic has been well explored in previous works [68, 69, 70, 71, 72, 73, 74, 75]. It is shown in these works that how FPGA-based bit-serial arithmetic operations offer advantages in area utilization while suffering from performance degradation. Our propose design shows how replication of bit-serial operations in a SIMD architecture can provide performance and area improvement by packing more processing elements in a 2-D processor array. In this regard, some of the previous works propose bit-serial ALUs and then replicate them to form a processor array [76, 70, 68]. These works show that with significant concurrency, serial arithmetic can counterintuitively improve throughput compared to bit-parallel implementations. It is discussed in [70, 68] that despite the decreased throughput of individual serial operators, replication of serial arithmetic can provide an increase in throughput compared to bit-parallel pipelines. Compared to previous works, our bit-serial ALU is more resource-efficient so that we only utilize 2 LUTs and 2 FFs, while for example, our ALU in [76] takes 16 LUTs and 7 FFs without supporting the multiplication operation. The works presented in [77, 78] use bit-serial arithmetic to occupy minimum amount of hardware resources for on-line encryption using bit-serial XOR and addition operations. Despite our work, their multiplier is not fully implemented using bit-serial. The use of bit-serial multiplication in embedded systems has been well studied [79, 80, 71, 85, 83, 86, 76, 81, 69]. While in these works, the standard multiply-accumulate method is implemented, we have developed a bit-serial radix-4 Booth's multiplication. Similar to [70, 68], we use Booth's multiplication [82], but with optimizations on the logical shift operations and also not using any shift register in the ALUs. This saves some resources within the ALUs and therefore, increases the level of concurrency by packing more PEs in the design. The bit-serial and bit-sliced methods have also been utilized for implementing the arithmetic operations of machine learning applications
[83, 84, 85, 51, 86, 87, 88]. These research works are primarily driven by the observation that using the bit-serial operations, the data-width required for ML applications can easily vary across

networks and the layers of the same network. For example, in [83, 84], bit-serial is used to match the data-width of individual neural network layers for CNNs. The work in [51] uses a serial processor unit that processes one bit at a time and is width-specific for each group of weights with various bit-widths. The [86] method relies on bit-serial compute units and the parallelism that is naturally present within neural networks to improve performance and energy. In [87], bit-serial operations are utilized to improve the memory bandwidth utilization and being capable of retrieving any-precision data from a compact memory storage. The difference between their work and our method is using the standard multiply-accumulate multiplication instead of our improved Booth's multiplication, as well as utilizing DSPs versus our method that is LUT-based. In [88], similar to our work, they present hardware-efficient MVM implementation techniques using bit-serial arithmetic, but at a lower operating frequency and based on a custom design just for CNNs, not as an overlay. Bit-sliced method has been implemented in [85] for multiplication operation in CNN networks. However, despite our method that the slice sizes can vary from 2 to 32 bits, in [85] the slice size is fixed. Therefore, our method is distinguished from the discussed previous research works as it supports all types of neural networks and is programmable at run time.

Table 1: Comparing the Previous FPGA-based Overlay Implementations

Ref.	**Benchmarks**	**FPGA**	**Data Format**	**Freq. (MHz)**	**Code**	**Method**
[22]	MLP	Zynq U	FxP 18	-	HDL	A flexible architecture that fully exploits DSP blocks
[14]	MLP	Zedboard	FxP 13	100	C++/HLS	Linear array of PEs reprogrammed at run time
[15]	MLP	ZYNQ-7	Integer	300	Verilog	C program on ARM, Quantized to power-of-two
[16]	CNN	Stratix V	FxP 32	200	RTL generator	A number of PEs and memory blocks, The design is translated to RTL, manually or automated
[17]	CNN	Kintex U[a]	FxP 8, 16	200	HLS	An automation flow directly compiles high-level ML definitions to the final FPGA accelerator
[18]	CNN	Arria 10	FlP 8-32	450	HDL	A domain specific VLIW compilers ML to target overlay
[19]	CNN	KU115	FxP 8, 16	200	HDL generator	A pipelined parameterized process engine Design flow from ML frameworks to board-level FPGA
[20]	CNN	Kintex-7	FxP 8	200	HDL	A compiler formulates the ML computation and maps the network into hardware
[21]	CNN	Virtex-7 Virtex U	FxP 16	650	HDL	A highly-scalable overlay compilation framework optimized for the tiled structure of FPGAs

[23]	CNN	Kintex-7	FxP 8	200	HDL generator	A domain-specific overlay processor automatically compiled into executable codes
[45]	CNN LSTM	Stratix-V	FxP 8-32	150	HDL generator	Generates hardware implementations with RTL-HLS hybrid templates
[64]	MIPS	Various[2]	FxP 16	50	RTL	Portable custom instruction set extensions, Mapping the overlay interconnection into the FPGA's switches
[65]	Signal Processing	Virtex-5	FxP 24	196	HDL	CGRA array (256 PEs) with a soft-core processor for controller
[13]	Image Processing	Stratix III	FxP 32	-	HDL	Array of PEs interconnected by programmable switches configured by the soft processor at run-time
[66]	Image Processing	Arria 10	FxP 8-32	200	C++	Heterogeneous PEs for threads maps the dataflow kernels onto the accelerator
[67]	MCNC[3]	Virtex 5 Stratix IV	-	-	HDL	A cross-compatible embedded system LUT-based FPGA architecture

[1]U means Ultra.

[2]Including StratixIV, Spartan-6, Cyclone-II, and Spartan-3.

[3]Benchmark netlists for floorplanning and placement.

From different machine learning architectures that can run on the processor array, LSTM networks have recently been more of an interest for time-series applications. These networks are expensive in computation and communication operations. Therefore, providing a low-latency design is well studied using FPGAs. The work in [100] implements a compact and configurable model in which the scale and size of the neural networks are configurable using fully-pipelined hardware. Several configurable buffers are used to fully utilize the bandwidth of the external memory used for storing the network parameters. The other work in [42] implements a single-threaded SIMD ISA paired with a distributed microarchitecture capable of dispatching over 7M operations from a single instruction. Their method exploits parallelism and parametrizability. It also benefits from the flexibility of running on different FPGAs. The pipelining is optimized using direct producer-consumer dataflow routing to reduce pipeline bubbles. The other optimization method is used on the LSTM networks is weight pruning, which is in [101]. Their ML network is pruned to create structured sparse features for the hardware-friendly purpose by using permuted block diagonal. They also utilized normalized quantization on the network parameters and mask matrices. The method in [102] utilizes pipeline and parallelism methods for the forward computing process. They also utilized weight pruning on 90% of the connections between the input layer and the hidden layer to generate a sparse LSTM network. The method in [103] is also using a structured pruning method to eliminate the imbalance computation, and irregular memory accesses so that only $\frac{1}{8}$ of parameters are reserved. In [103], they are pruning the entire columns of redundant weights. If the sum of the absolute value of weights in the same column is smaller than a threshold, then the whole column weights are pruned away. Using the FIFO strategy to transfer weights of different layers, the computations of several layers are put together. The computation between adjacent layers is also pipelined. In another method [104], they exploit the LSTM networks inherent parallelism so that the matrix-vector multiplication operation is performed in parallel in linear time. The other method in [105] uses stochastic computing along with stochastic memories to simplify the fundamental arithmetic circuits. In their stochastic memory, the primary objective is to convert binary data into

stochastic values then stochastic values are fed to the network. The method in [53] discusses that the random nature of the pruning technique transforms the dense matrices of the model to highly unstructured sparse ones, which leads to unbalanced computation and irregular memory accesses and thus hurts the overall performance and energy efficiency. Therefore, they use a structured compression technique that could reduce the LSTM model size and eliminate the irregularities of computation and memory accesses. Their method employs block-circulant instead of sparse matrices to compress weight matrices and reduce the storage. They proposed to break down the original single pipeline into several smaller coarse-grained pipelines and overlap their execution time by inserting double-buffers for each concatenated pipeline pair. All the weight matrices are compressed small enough to be stored in on-chip BRAM buffers instead of off-chip DDR memory. This reduces the execution time to the order of μs. In [106], the bit-sliced method is conducted to cascade enough slices for an optimum performance depending on the problem size. Slices are arranged into an n-dimensional structure for the adder to concurrent with the rest of the hardware in a pipeline fashion. In the [107] work, as irregular computation and memory accesses in unrestricted sparse LSTM limit the realizable parallelism, they use bank-balanced sparsity, a novel sparsity pattern that can maintain model accuracy at a high sparsity level while still enable an efficient FPGA implementation each matrix row is split into multiple equal-sized banks, and each bank has the same number of non-zero values so that it prunes the smallest 50% of weights. The same method is utilized in [52], in which a load-balance-aware pruning method is used to generate sparse LSTM to 90% and the execution time is in order of μs for [107, 52]. In [108], in-depth investigation of precision versus accuracy using a fully hardware-aware training flow is performed. During training, quantization of all aspects of the network including weights, input, output, and in-memory cell activation are taken into consideration. Their method avoids the use of redundant high-precision calculations for on-chip dense models and implements coarse-grain and fine-grain parallelism. The method in [109], develops a low-latency real-time implementations for pipelining and loop unrolling. In this method, instead of off-chip memory, a custom on-chip memory paging/controller system is implemented that efficiently supplies the needed memory

values to the module. In [110], compression is conducted to the number of parameters by 7×, and it stores the two non-linear functions approximation in a single-port ROM. In [33], the inner-most loops are unrolled, and computation in parallel to minimize latency is performed. They also use pipelining and compression techniques. They reshape the parameter matrices to ensure that they can be accessed sequentially for the tiled computation. In [111], they also use parallelism and complexity reduction through precision. The other method in [112] uses pipeline methods to parallelize the forward computing process. To optimize their implementation, they also use multiple methods, including tiled matrix-vector multiplication, binary adder tree, and overlapping the computation and data access. In [113], scalable division method is proposed. The size of the target LSTM and the number of boards used in the prototype can be freely changed. They develop a large-scale AI system called F1ow-in-Cloud. The current prototype is consisting of multiple FPGA boards with a high communication bandwidth network. In [114], low-power and high-speed features that are achieved through overlapping the timing of the operations and pipelining the datapath. The method in [36] involves designing a parallel multiply-accumulate unit configuration to perform the matrix-vector multiplication. In [**?**], a comparative study of FPGA, GPU, and FPGA+ASIC in-package solutions for integrating an ASIC chipset and TensorRAM, with an FPGA as system-in-package is provided to enhance on-chip memory capacity and bandwidth. Their method also provides compute throughput matching the required bandwidth. A summary of LSTM custom accelerators is presented in Table 2.

Table 2: Comparing the Previous FPGA-based LSTM Implementations

Ref.	Net Size[a]	Data Format	Freq (MHz)	Delay (ms)	LUTs (#)	FFs (#)	DSPs (#)	BRAMs (#)	GOPS	Power (W)	FPGA	Method
[100]	1024	FxP 16	238	-	3092	1703	19	16	7.64	0.88	Zynq-U[b]	HDL
[?]	256	FlP 11	250	0.42	845719	-	5245	8192	370	125	Stratix-10	HDL
[101]	200	FxP 8	150	-	-	-	-	-	1850	1.67	Arria-10	HDL
[102]	256	FlP 32	-	-	-	-	-	-	-	2.07	Zedboard	HDL
[103]	512	FxP 16	220	-	120248	-	160	1238	339.7	-	Stratix-V	HLS
[104]	128	FxP 18	154.3	0.002	125448	111118	512	-	4.534	1.51	Virtex-7	HDL
[105]	16	FlP 32	100	18.58	9529	8456	0	0	-	0.072	Zynq-7	HDL
[53]	1024	FxP 16	200	0.01	249423	390719	2646	946	-	22	Kintex-U	HLS
[106]	-	FxP 16	254	-	-	-	229	-	-	-	Virtex-U	HDL
[107]	1024	FxP 16	200	0.002	289000	-	1518	2509	304.1	19.1	Arria-10	HDL
[52]	1024	FxP 16	200	0.08	293920	453068	1504	947	282	41	Kintex-U	HDL
[108]	128	FxP 8	266	-	55000	-	-	170	1833	0.87	Zynq-U	HLS
[109]	10k	FxP 12	100	0.35	177432	191117	389	521	-	-	Zynq-U	HLS
[110]	600	FxP 8	100	9.9	6785	3591	34	24	-	-	Zedboard	HDL

[?]	250	FlP 32	150	390	189871	181634	1176	112	7.26	19.63	Virtex-7	HLS
[111]	100	FxP 5	142	1170	161574	51213	195	339	308.05	-	Zynq-7	HLS
[112]	128	FlP 32	-	-	46834	26212	200	264	-	2.462	Zedboard	HDL
[113]	256	FxP 16	100	0.01	103825	46329	512	384	101.05	14.1	Kintex-U	HLS
[114]	64	FxP 16	164	0.09	12714	10640	38	-	7.51	0.28	Zynq-7	HDL
[?]	128	FxP 16	142	0.90	7201	12960	50	16	0.264	1.942	Zynq-7	HDL
[?]	1024	FxP 8	275	3.13	567982	-	4880	9018	10600	28.3	Stratix-10	HDL

[a]Number of hidden nodes in the LSTM layer.

[b]U means Ultra.

In addition to the LSTM networks, MLP networks are another type of communication-bound ML applications. A summary of the previous works on FPGA-based MLP implementation is presented in Table 3. This table shows that most of the previous works are customized for a specific MLP network architecture with a fixed data-width. However, our overlay supports any MLP architecture in any data-width. The previously studied MLPs are mostly small and the execution time is in order of *mu*s. It is also shown that most of the previous works focus on fixed-point data type rather than floating-point. The blank spaces in this table are the information that is not reported in the previous works. From these studies, the FPGA-based overlays include [14, 89, 46, 15, 22] and they are run-time programmable for only MLP architectures, not all ML applications. The work presented in [14] proposes a co-processor that is configurable at run-time and allows application developers to modify the MLP network parameters and eliminates the need to resynthesize. The other method that supports configurability at run-time is [89]. This study uses a software interface to generate the MLP networks and provide a high-performance hardware design. The other MLP overlay is [46] that proposes a framework using a flexible heterogeneous streaming architecture for building binariezed MLP accelerators. Their design provide the fastest classification rates reported on the utilized image classification benchmarks. The [15] study introduces a high-performance MLP accelerator overlay using a multiplication-free design. In their design the network parameters are quantized to power-of-two values, that replaces the multiplication operations with logical shift operations. Another overlay design for MLP networks is [22] that has optimized the architecture and the overlay interconnect around the DSP blocks. From the previous works presented in Table 3, in addition to accelerating the inference phase of ML algorithms, some of them have implemented the training phase as well [94, 96]. In [94], the authors propose a training method based on descendent gradient. Their design performs the operations in serial since they aim to reduce the resource utilization. In [96], instead of classic back-propagation methods, the training is implemented using weight perturbation techniques. The [91, 92, 97, 99] studies perform a design space exploration to find the optimum design for accelerating MLP inference. The [90, 93, 95, 98] designs are optimized for a specific application.

The [90] proposes an MLP for activity classification, [93] proposes an MLP for a primitive gas recognition system for discriminating between industrial gas species, [95] is designed for real-time cancer detection, and [98] detects anomalies in ECG signals.

Table 3: Comparing the Previous FPGA-based MLP Implementations

Ref.	**Network**	**Exe. Time**	**Data Width**	**Freq. (MHz)**	**FPGA**
[14]	(64, 16, 64)	83.66 μs	FlP 32	100	Zedboard
[89]	(100, 9, 2)	31.04 μs	FxP 32	33	Virtex-6
[90]	(7, 6, 5)	270 ns	FxP 16	100	Artix-7
[46]	(1024, 1024, 1024)	2.44 μs	FxP 1	200	Zynq-7000
[91]	(29, 10, 6)	7.81 μs	FxP 32	100	Spartan-6
[15]				300	Zynq-7000
[92]			FlP 32	250	Arria 10
[93]	(8, 4, 5)		FxP 16	10	XC4000E
[94]	(2, 5, 2, 1)	1.96 ms	FxP 16	50	Spartan-6
[22]	(11, 12, 10, 3)		FxP 18		Zynq Ultra
[95]	(15154, 512, 512, 2)	10 μs	FxP 8	295	Arria 10
[96]	(32, 16, 8, 16, 32)		FxP 12		Kintex-7
[97]	(784, 256, 256, 10)		FxP 8	295	Arria 10
[98]	(8, 4, 1)		FxP 16		Zynq
[99]			FxP 32	150	Virtex-7

An optimization on our processor array overlay is developed to in addition of inference acceleration, support online-training for the MLP networks. We would discuss the previous works

in this area and compare it with our overlay design. A summary of these previous works has been presented in Table 4. In this table, the [115] has developed a new back-propagation method using ternary operations on MNIST dataset with batch-size of one. The [116] study has implemented a systolic array for MLP networks with pipeline method for back-propagation algorithm. They also share the hardware resources for both inference and training phases. The other study in [96] proposes an optimized implementation of online ML training based on weight perturbation. They use ping-pong buffering for the network parameters and unroll the network computations on FPGAs using a fully-pipelined method for MLP training. The [117] implements on-chip back-propagation training algorithm to implement a small MLP network. In [118], the Quasi-Newton method is used for neural network training on a Virtex-7 FPGA. Another study in [124] also uses Quasi-Newton method for training. In this method, an inexact line search method is implemented to replace the exact line search method. The [119] study uses online training method to implement a back-propagation MLP training on a Virtex-6 FPGA. Parallel-pipeline structures are utilized to accelerate the computational process. The [120] evaluates the effects of arithmetic precision on hardware implementation of MLP neural network training using matrix-vector multiplication operations. Another study in [121] uses a generic high-speed integrated circuit to experiment with a large number of formats and designs in MLP back-propagation algorithm. The other work in [122] introduces hyper-dimensional computing as an alternative computing paradigm for developing efficient and robust MLP training algorithm. The [123] proposes a new neuron representation using time-division multiplexing for DSPs. The [125] study accelerates the training latency by using hardware-oriented algorithmic optimizations. They also remove dependencies and expose parallelism methods for transforming the algorithmic structures. In all these previous works, the design is customized for a specific small benchmark. However, in our overlay design, different sizes of MLPs can be implemented without the need to re-synthesize. Additionally, the data-width can vary based on the application's requirements.

Table 4: Comparing the Previous FPGA-based MLP Training Implementations

Ref.	**Network**	**Data Width**	**Latency**	**Freq. (MHz)**	**FPGA**	**Method**
[115]	(784, 128, 10)	FxP 3			MAX10	Quartus
[116]	(2, 6, 3, 2)	FxP 16	0.7 μs	10	Virtex-4	HDL
[96]	(4, 7, 12, 3)	FxP 14	171 μs		Kintex-7	HLS
[117]	(2, 2, l)	FlP 32	10 μs		Spartan 3	HDL
[118]	(8, 15, 8)	FlP 32	41 *m*s	250	Virtex-7	HDL
[119]	(15, 32, 2)		2.95 μs		Virtex-6	
[120]		FxP 16		33	XC4000	HDL
[121]	(10, 10, 10)	FxP 18		200	Virtex-2	HDL
[122]	(617, 26)	FxP 16	1 *m*s	200	Kintex-7	HDL
[123]	(60, 15, 10, 5)	FxP 16		190	Virtex-5	
[124]	(15, 32, 1)	FlP 32	5.5 *m*s	250	SUME	HDL
[125]	(46, 15, 1)		37.9 *m*s	300	ZCU102	HDL

Chapter 3

Proposed Approach

Our design was aimed to implement an FPGA overlay as a 2-D processor array, which provides programmability and reusability of the underlying fabric for implementing machine learning algorithms. Machine learning algorithms are continually evolving which require updates to a wide variety of IoT FPGA edge devices. Having to recode the hardware design for each update and resynthesize for each FPGA device further proliferates the historical limitation for the current FPGA-based designs. The limitations including the low productivity, lack of portability and reuse, and the need to understand hardware design was addressed using our method in this dissertation. We focused on evaluating the most challenging FPGA-based design goal that how to bring software levels of productivity, portability and reuse to rapdily changing ML networks deployed throughout a growing number of different IoT FPGA edge devices. On the other hand, when implementing FPGA-based ML networks acceleration, data access latencies within an FPGA requires new architecture approaches that increase the storage efficiency of the limited capacity on-chip BRAMs and eliminates serialization of weight transfers and partial results between the multiply-accumulate array and the BRAMs. Ideally, all ALUs would be provided full concurrent access to weights and partial results. This was an important design goal of our processor array presented in this dissertation. Our overlay is a Single-Instruction-Multiple-Data (SIMD) processor array designed to support ML applications running in IoT FPGA edge devices. This overlay is a configurable overlay that can be sized for different FPGAs. As an overlay, it is programmable and executes the Microblaze instruction set architecture (ISA). This allows rapid updates and brings code portability across different FPGA devices. The overlay architecture accelerates the ML algorithms by implementing the soft processor instructions on hardware. In our processor array, computation is carried out by the processing elements connected using four-neighbor connectivity to form a 2-D array of PEs. The 2-D array of PEs implement a SIMD architecture. In a SIMD architecture, at each time, the same instruction is conducted on multiple

data. In our design, the operations are performed in parallel in all processing elements. Each PE has its own register file and the same operation is implemented on each PEs register file. The simple processing elements are made of an ALU for arithmetic operations and a register file for data storage. The performance benefits of hardware accelerators desired to have efficient ALUs in the processing elements. Therefore, we explored using bit-serial operations for implementing the arithmetic operations in ALUs. The bit-serial SIMD processor arrays are utilized to use the large memory bandwidth more efficiently within a memory chip. This is provided by performing a significant number of massively parallel bit-serial computations and thus achieving high performance in ML inference acceleration. The register file is implemented using distributed BRAMs in a memory-centric method to best support both the communication and computation bound ML algorithms [24].

3.1 System Architecture

In contrast to projects developing point designs for a particular neural network configuration, our method was developed to fill the need for a generalized reprogrammable solution. This architecture addresses the source of communication and computation latencies found across various machine learning algorithms. As an overlay, it is a step towards enabling the portability of code over different FPGAs. Figure 2 shows the 2-D processor array that includes the tiles and PE-blocks. Each PE-block consists of the processing elements that perform the arithmetic operations. Operationally, a master processor (MicroBlaze in Figure 2) sequences the running of instructions. The algorithms that run on the processor array are programmed using the overlay's ISA on the MicroBlaze and are executed one by one. The instructions go through the AXI slave registers to be sent to the overlay. The controller reads the instructions from the AXI slave register and runs it on the processor array. As typical with SIMD architectures, programs contain data-parallel instructions for the processor array and control instructions for the master processor. Instructions for the 2-D processor array are placed into the slave register, and the master processor is put on sleep to maintain proper instruction sequencing. This decouples the execution of

multicycle instructions in the processor array from the Microblaze. All arithmetic and data movement instructions are executed on the processor array. This simple protocol allows the processor array to compute a set of instructions decoupled from the master processor. Although our design adopted the standard MicroBlaze's ISA for convenience, it has been designed to easily support other processor's ISAs as well. The MicroBlaze ISA is implemented in our design to allow utilizing the standard MicroBlaze compiler for a direct compilation from C/C++ codes to the overlays instruction set.

Figure 2 shows an overview of the processor array architecture that contains a 2-D SIMD array of $m \times m$ tiles. Each tile contains $n \times n$ PE-blocks. The number of tiles and PE-blocks within a tile is configurable to allow the SIMD processor array size to be tuned to any specific FPGA. Each PE-block consists of $l \times l$ PEs that share a BRAM for register storage. Each PE has a bit-serial ALU. In designing and implementing the ALUs, an initial design trade-off was made to investigate Look-Up-Table-based (LUT-based) bit-serial arithmetic circuits as opposed to full-precision bit-parallel Digital Signal Processing (DSP) units to increase the density of PE units that could be packed within an FPGA. Therefore, to pack more PEs within the FPGA, no DSPs are used in our processor array. This is a significant difference between our method and HLS driven designs. While DSPs provide reduced latency for full-precision operations, they can limit concurrency and result in inefficient resource utilization for less than full-precision operations. For example, experiments on a Virtex-7 FPGA for a 32-bit design show that utilizing DSPs for arithmetic operations in ALUs limits the number of PEs to only 676, whereas using bit-serial ALUs results in up to $16k$ PEs on the same FPGA. Therefore, the additional cycles resulting from bit-serial operations are amortized through parallelism resulted from SIMD concurrent operations in PEs. This approach allowed a more efficient utilization of available BRAMs to reduce the inference latency associated with weight stall cycles while increasing the density of PEs within the FPGA. In the next sections, each part of the processor array hardware design is separately discussed in more detail.

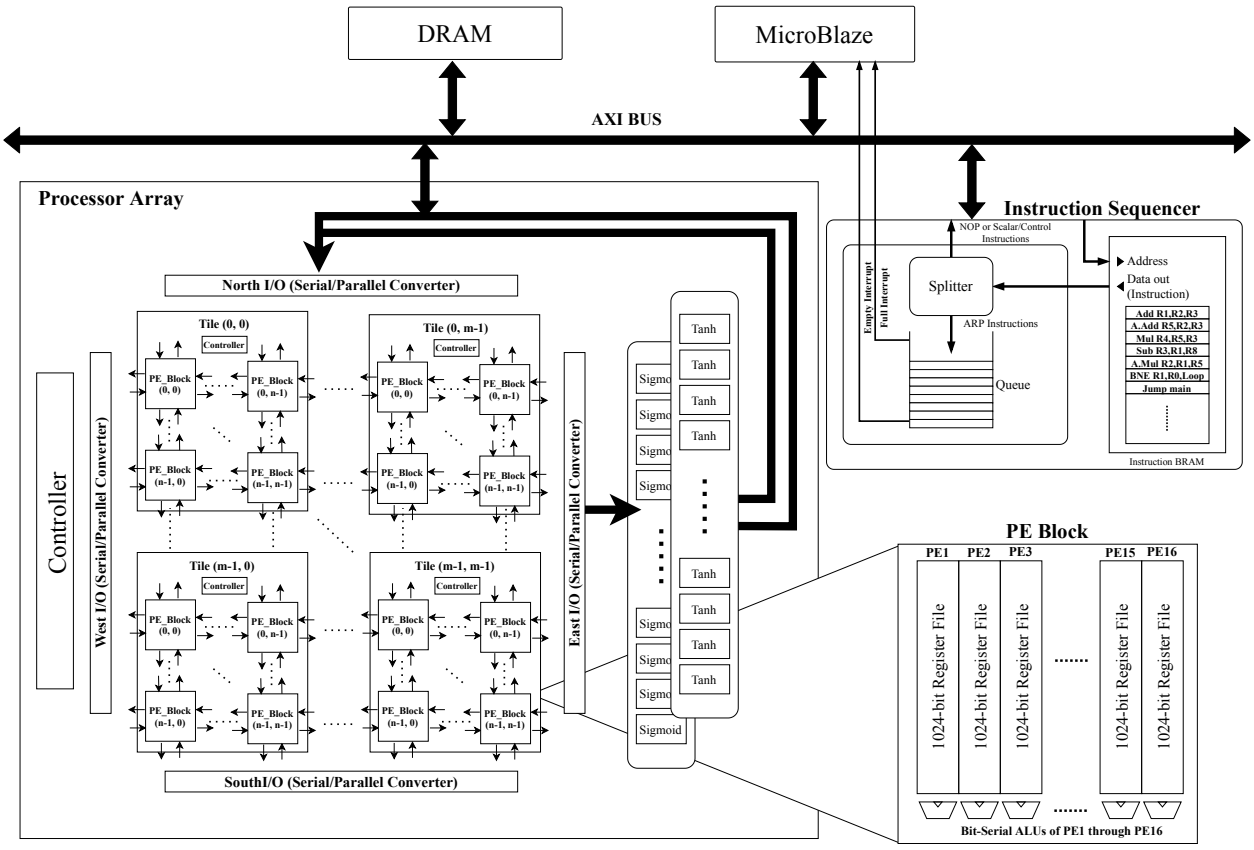

Figure 2: Overlay architecture.

3.2 ISA

Our overlay implements a subset of MicroBlaze instructions as listed in Table 5. Arithmetic instructions include addition (*add*), subtraction (*sub*), multiplication (*mult*), and *relu*. Internal data movement instructions (*Move*) are provided as a linkable library. In the processor array, *Move* instructions simply direct the flow of data between different PEs. The arithmetic instructions take two input registers as input operands and one output register for storing the result of the arithmetic operation. For example, as also described in Table 5, the *add* R_d, R_{s1}, R_{s2} instruction add the values stored in R_{s1} and R_{s2} registers and stores the result in R_d. Because of the SIMD architecture of the design, this *add* instruction is performed on the same registers of all PEs, but each PE's registers store different values in their register file. The same applies for other arithmetic instructions of *sub* and *mult*. The *relu* instruction, which is used for CNN networks,

takes one input register, the function is applied to that value and the output is stored in R_d. The *NEWS Moves* instructions are used for internal data movement between the PEs. *Move_N*, *Move_E*, *Move_W*, and *Move_S* instructions move the data from the source (R_{s1}) register to the destination register (R_d) of their adjacent PE in their North, East, South, and West side. In the original design, each PE is only connected to its 4 adjacent PEs. Therefore, the R_{s2} register is not used. However, as discussed in Section 5.1, the original processor array is then optimized for internal data movements. In this case, since each PE is connected to more than one PE in different distances, the R_{s2} register determines the PEs distance. For example, *Move_E* $R1$, $R2$, 4 moves the values of $R2$ of all PEs to the PEs that are in the 4_{th} east column from the original PE.

To allow easier implementing of different applications on the processor array, some domain-specific SIMD macros are implemented in software using the described basic ISA. Table 5 also shows these software macros. All these software macros are implemented using the basic ISA (top instructions in Table 5). More domain-specific macros can easily be implemented using these functions and instructions as needed. From the presented software macros in Table 5, matrix-vector multiplication (*MVM*), vector-vector element-wise addition and multiplication (*VVA*, *VVM*) are used for MLP/LSTM/GRU networks. For the CNN networks, 2-D convolution (*2D_conv*), 2-D padding (*2D_pad*), 2-D average-pooling (*2D_avg_pool*), and 2-D max-pooling (*2D_max_pool*) are called. In the *MVM* software macro, the input registers determine the registers that store the input matrix and the input vector and R_d is the output vector. The input values are stored in the R_{s1} and R_{s2} registers before calling this software macro. The *VVA* and *VVM* software macros also take R_{s1} and R_{s2} as the input vector and the result is stored in R_d. The same applies to the *2D_conv*, *2D_pad*, *2D_avg_pool*, and *2D_max_pool* software macros. The input matrix is in R_{s1} and the output matrix is stored in R_d register. The other required parameters such as kernel size and stride size are also passed to the software macro, but are not mentioned in Table 5. More detail on the implementation of these software macros provided in Section 3.10.

3.3 Memory-Centric Tiles and PE-blocks

The processor array shown in Figure 2 appears as a standard 2-D processor array composed of tiles and PE-blocks. As shown in Figure 3, each PE-block forms a PIM computing component. Each PIM PE-block contains a $4 \times 4 = 16$ configuration of PEs with a local storage modeled as a traditional register file using the Xilinx RAMB18E1 BRAM block. The RAMB18E1 module has a depth of 1024 bits and a width of 16 bits, which resembles an array of 1024 rows and 16 columns of bits. In full-precision designs, operands are stored in a row-major format, where each row represents one register. This prevents concurrent access to the BRAM shared by multiple PEs. Conversely, in the PIM PE-block, operands for all 16 PEs are stored in a column-major format, i.e., vertically, as shown in Figure 3 (e). This configuration allows a single BRAM to provide concurrent accesses for all 16 PEs. By storing data vertically, each column of the BRAM (1024 bits) is dedicated to a specific PE (16 columns, one per PE). This means for 32-bit operands, the 1024 rows of bits are partitioned into 32, 32-bit registers per column. Register $R0$ of all 16 PEs occupy address range 0 to 31. Register $R1$ from address 32 to 63, and so on (illustrated in Figure 3 (e)). The number of registers and operand data-widths of a BRAM is configurable through simple addressing. In a design with N-bit data-width, the available 1024 column bits for each PE is accessed as $k = \frac{1024}{N}$ registers (shown in Figure 3 (e)). This allows a PE's 1024 bit storage to be viewed by the controller as $k = \frac{1024}{N}$ internal registers. For example, for 32-bit, 16-bit, and 8-bit data-widths (N), the number of storage registers (k) within the BRAM becomes $\frac{1024}{32} = 32$, $\frac{1024}{16} = 64$, and $\frac{1024}{8} = 128$, respectively. This ensured that all BRAMs are efficiently utilized, and all 16 PEs can simultaneously access their respective register files. The PE-blocks, thus, become the building blocks for a system with high data-level parallelism.

Figure 3 provides an expanded view of the PE-blocks within each tile. Figure 3 (a) shows the tiles and PE-blocks connected in a 2-D N-E-W-S interconnect. However, this network is virtual and shown only for illustrative purposes. Communications between PEs are implemented using reads and writes of data from one PE's register into another PE's register within the BRAM. This small processor array is an example design that includes 4 tiles and each tile includes 4 PE-blocks

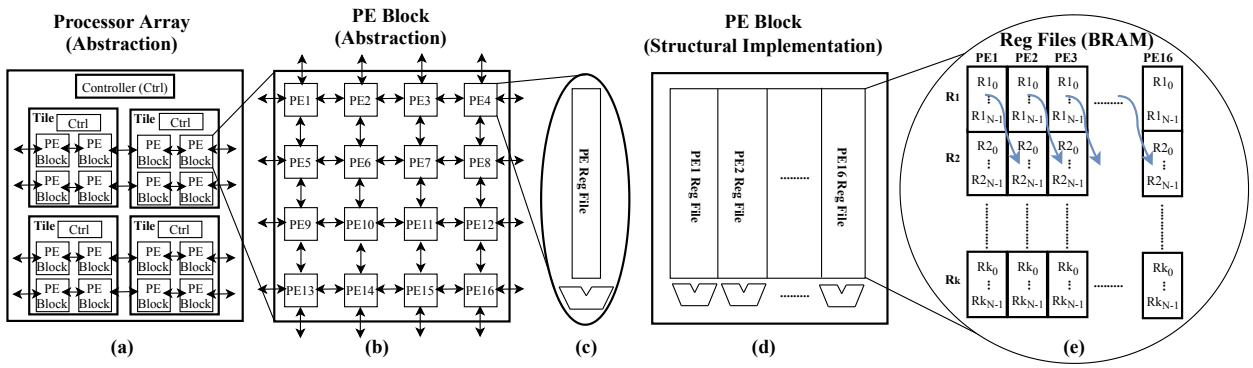

Figure 3: PIM Tile.

in Figure 3 (a). There is one controller inside each tile and a master controller for all tiles. Figure 3 (b) shows that a 4×4-PEs with nearest-neighbor connections comprise a PE-block. Figure 3 (c, d) show each PE contains a bit-serial ALU and a register file implemented within a column of a BRAM. Figure 3 (d) shows the structural view of a PE-block. The block that shows the PEs' register file is a RAMB18E1 BRAM block with 16 columns (one for each PE). There are 16 bit-serial ALUs inside each PE-block, each is connected to one column of the the BRAM block. Finally, Figure 3 (e) shows more detail on how a RAMB18E1 BRAM block provides shared register files for all 16 PEs in a PE-block.

Experiments showed that the optimal size of the PE-block is device-dependent. We identified two possible configurations for the PE-blocks by exploring different configurations of PEs with the Virtex-7 BRAMs: 1) connecting 16 PEs to a single 18Kb RAMB18E1 or 2) connecting 32 PEs to a single 36Kb RAMB36E1. The first option proved more efficient for the following two reasons: First, a PE-block of 16 PEs connected to a single BRAM can be abstracted to a square $4 \times 4 = 16$ PEs (Figure 3 (b)), rather than a rectangle of 4×8 or 2×16 block of 32 PEs. Utilizing the block of 16 PEs offers symmetry, simplifies interconnect, and allows for simpler control logic that deals with the four directions in a consistent manner. To illustrate, with a 4×4 PE-block, we can build any square 2-D processor array where its dimensions are multiples of 4, such as 4×4-PEs (1 PE-block), 4×8-PEs (2 PE-blocks), or 20×20-PEs (25 PE-blocks) processor arrays. Whereas, by using 32-PEs in a PE-block, one of the dimensions of the processor array

must be at least a multiple of 8, so we would not be able to produce a processor array with a size of 4×4-PEs, 12×12-PEs, or a 20×20-PEs. Second, as mentioned before, our design should create the smallest feasible PE-block to reserve the flexibility and configurability of the system. Additionally, our experiments showed that designs that utilize $18Kb$ RAMB18E1 versus $36Kb$ RAMB36E1 achieve higher operating frequencies with lower timing issues. Therefore, we utilized an $18Kb$ RAMB18E1 with 16 PEs inside each PE-block. The PE-blocks are of size 4×4 and can be arranged in different ways to build larger processor arrays.

3.4 ALUs

The 1-bit serial ALUs are used in the overlay. The goal of using bit-serial ALUs was to design the smallest possible ALU that efficiently utilizes available FPGA resources to allow packing a large number of those ALUs. The bit-serial ALUs are implemented using only 2 LUTs and 2 FFs. This simple ALU can support variable precision arithmetic operations. The bit-serial ALUs operate on 1 bit at a time. The controller iterates through the whole bits and sends only one bit to the ALUs at each time. When the operands data-width changes, it is just the number of iterations that changes and the ALUs bit-width does not need to be modified based on the arithmetic precision. Therefore, as each PE's register file is mapped vertically within a BRAM, the controller sets the bit iterations based on the data-width. This flexibility allows data-width changes to be made in software. The utilized data format in our design is two's complement. This data format handles any custom fixed-point data-width in various custom formats. Using this data format in the arithmetic operations, the same algorithms are executed on the integer and fraction parts of a number, regardless of the number of bits for each part. Shown in Figure 4, all ALUs perform the same operation (single-instruction) on parallel data (multiple-data) stored in their corresponding BRAM register files. The ALUs for addition/subtraction and radix-4 Booth's multiplication are designed to only perform addition or subtraction. The ALU's operation is determined using a control signal ($aluOp$) sent from the controller to switch between addition or subtraction operations. To perform the arithmetic operations, the controller first sets the input operands

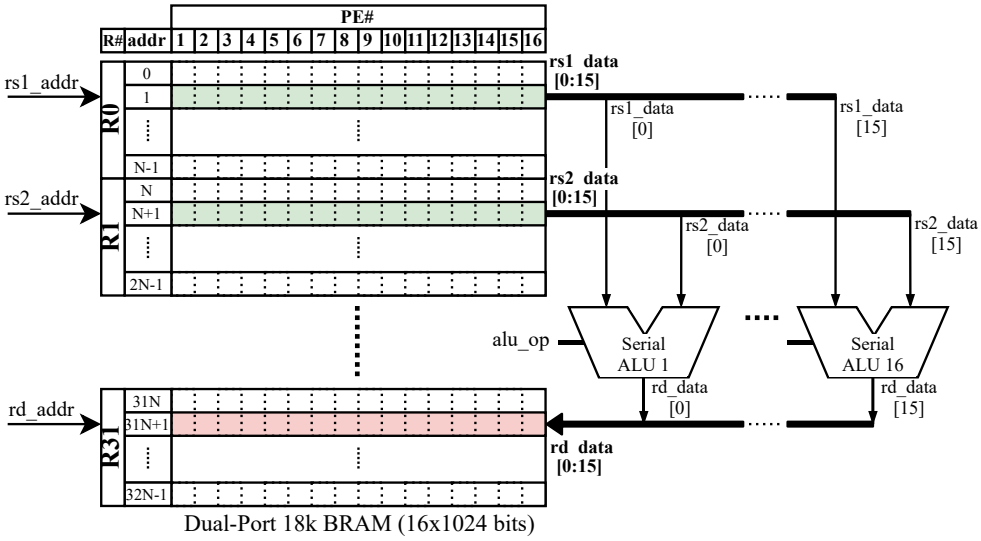

Figure 4: Bit-serial ALUs.

addresses ($rs1\_addr$, and $rs2\_addr$). A BRAM read operation will read from those addresses which are two rows of the dual-port BRAM block (highlighted in green in Figure 4). The values of those two rows are one bit of the input operands to the 16 ALUs (one bit per ALU). Then, the arithmetic operations are performed on those single bits in parallel in all 16 ALUs. The results is then stored in the destination register in BRAM (highlighted in red in Figure 4). The address of the destination register is determined by the controller (rd_addr). Then the controller again sets the address for the input operands next bit that should be processed. This iterations continues until all bits of the input operands are processed and the final result is stored in the BRAM.

3.5 Bit-serial Arithmetic

The ALU's structure is very simple. It has two 1-bit inputs for operands (rs_1, and rs_2), and one 1-bit output (rd) for the result. For the bit-serial addition/subtraction operation, the ALU also has a 1-bit register called cb to represent the $carry\_in$ and $carry\_out$ of a full-adder as well as the $borrow\_in$, and $borrow\_out$ of a full-subtractor. The cb register is updated each clock cycle according to (1). The rd output, which is the same for the adder and the subtractor, is represented in (2). For addition/subtraction operations, at each iteration, the controller assigns the addresses

of rs_1 and rs_2, the data is read from BRAM, the arithmetic operations based on (2) are performed, the carry out and borrow out registers are set based on (1), and the result is stored in *rd* in BRAM based the address that the controller has set. Assuming data-width of N bits, the total clock cycles spend on add/sub instructions is $2N$. It takes N clock cycles to read the two input operands from the register files and perform the operation on them (one bit at a time), and it takes N clock cycles to write the result into the register file (one bit at a time). It should be noted that since a dual-port BRAM block is used, the two input operands are read at the same iteration, each using a separate port of the dual-port BRAM.

$$cb \Leftarrow \begin{cases} rs_1 \cdot rs_2 + rs_1 \cdot cb + rs_2 \cdot cb, & aluOp = ADD \\ \overline{rs_1} \cdot rs_2 + \overline{rs_1} \cdot cb + rs_2 \cdot cb, & aluOp = SUB \end{cases} \tag{1}$$

$$rd \Leftarrow rs_1 \oplus rs_2 \oplus cb \tag{2}$$

The bit-serial *Move* instructions are implemented using two bit at each iteration. The bits of the source register are read from the BRAM register file and are written into the BRAM destination register. The controller sets the addresses for the source and destination registers. Using a dual-port BRAM, assuming data-width is N, it takes $\frac{N}{2}$ clock cycles to read N bits of the source register (two bits at a clock cycle using the dual-port BRAM), and it takes $\frac{N}{2}$ clock cycles to write those N bits into the destination register. Therefore, the total cycles per *Move* instructions is $\frac{N}{2} + \frac{N}{2} = N$ clock cycles. The bit-serial multiplication is implemented using radix-4 Booth's algorithm [82]. In our method, we have optimized this multiplication algorithm by eliminating the algorithms' logical shift operations by just modifying the operands addressing. We now describe how our improved Booth's radix-4 algorithm is implemented. Typical implementations of $N \times N$-bit Booth's radix-4 multiplier circuits include a $2N$-bit wide shift register in which the partial product (P) is processed using the Multiplicand (M). At each iteration of the algorithm, $0M$, $1M$, or $2M$ is added or subtracted from the higher N bits of the product register. The higher N bits are then shifted right towards the lower end of the product register. Figure 5 shows how we

were able to eliminate the shift register and control circuitry. Figure 5 starts the first iteration on the two least-significant bits of the Multiplier (R) by reading two rows of all BRAMs at the same time. Each PE-block has 16, 3-bit registers (shown as $Q1$ for $PE1$ in Figure 5) to hold the operation (addition or subtraction) that each ALU will perform. Once the operation is determined, the controller sets up the BRAM addresses to output the bits of the product P ($R1$ and $R2$ registers) starting with the least significant bits and the bits of the Multiplicand M ($R4$). The results of adding or subtracting those bits are then stored back in place of the product bits. When subtracting $2M$, a 0 is provided as the first bit, assimilating the behavior of a logical shift right operation. The controller then increments the product register pointer (rd_addr arrows in Figure 5) by two, in preparation for the next iteration. Finally, the controller sets the BRAM control signals to double sign extend the result in preparation for the next iteration where the operation (addition or subtraction) happens two bits further from the previous iteration. At that point, the first iteration is completed ($Iter\#1$ in Figure 5), and the two least significant bits in the product registers do not change going forward. The same sequence occurs for the next iterations until the multiplication is completed. For an $N \times N$-bit radix-4 multiplication, the algorithm takes $\frac{N}{2}$ iterations to complete. Each iteration takes $2N$ cycles for addition/subtraction and 4 clock cycles for the sign extend operations for the radix-4 algorithm. This results in a total of $N^2 + 2N$ clock cycles for the bit-serial Booth's radix-4 multiplication. If the bit sequence is "000" or "111" for all 16 PEs, the controller skips performing the operation saving $2N$ clock cycles. Using this method, in the best case when all inputs are "0" or "1", $\frac{N}{2} \times 2N = N^2$ clock cycles are saved in the radix-4 algorithm.

3.6 Controller

The controller is implemented as a finite state machine that takes a standard 32-bit custom instruction to generate control signals of different parts of the system. It sets the op_codes of the ALUs, the ALU's operands addresses, and the write-enable signals of BRAMs for each instruction. For arithmetic instructions, the controller issues op_codes for the ALUs (add/sub)

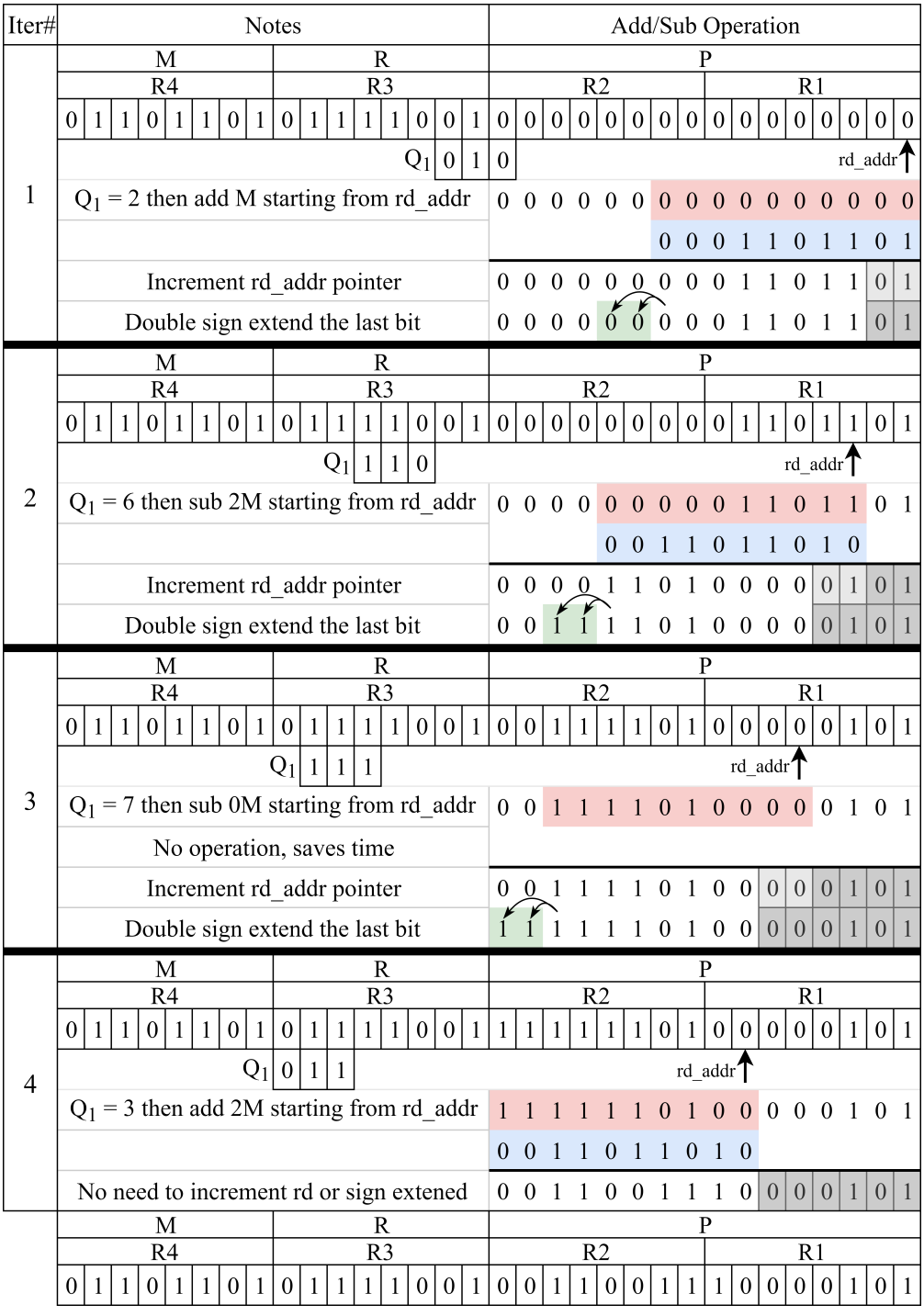

Figure 5: Bit-serial Booth's Multiplication Algorithm.

and sets the data addresses for reading and writing the operands in the BRAMs. Therefore, for arithmetic operations, the controller sends the right data from the register files to the ALUs at the right time. For internal data movement instructions (*NEWS Moves*), the controller sequences a series of microoperations to BRAMs to move data within one BRAM or between BRAMs of different PIM PE-blocks. Because of using serial arithmetic and serial data movements, the controller needs to send a series of BRAM and ALU control signals to achieve the correct functionality. Therefore, the controller has registers (rs_1_ptr, rs_2_ptr, rd_ptr) that store the pointers (addresses) of source and destination data at a certain time. For the activation functions, the controller sets the select bit of a multiplexer to choose between the *Sigmoid* and the *Tanh* modules. As shown in Figure 2, each tile has its own controller that is synchronized with the other controllers in the design. The controller within each tile handles the operations of PE-blocks inside that tile, and the master controller (left side of Figure 2), handles the operations between the tiles. The connection between the soft processor (MicroBlaze) and the controller is implemented using the AXI slave registers of the overlay's IP. The MicroBlaze writes the instructions into the AXI slave register and the controller reads from that slave register, decodes the 32-bit instructions, and sets the BRAM addresses using its FSM-based design.

3.7 Data Movement

As disscussed, the PEs of each PE-block are connected to their 4 adjacent PEs in *NEWS* sides. The same connections exist for the PEs on the edges of PE-blocks to connect two PE-blocks and two tiles together. Figure 3 (b) represents these connections between PEs in a PE-block and to the adjacent PE-blocks. For the PEs on the edges of the 2-D processor array, the data can input and output from the *NEWS* directions into the I/O buffers. This design implements the *Move_N*, *Move_E*, *Move_W*, and *Move_S* instructions for transferring the data between the PEs. Since the local data of each PE is stored in one column of a RAMB18E1 block, to move the data from a specific register of a PE into another register of another PE, we need to read the data from that specific location of the BRAM and store it in another location. Therefore, the *Move* instructions

are nothing but reading and writing the data from/into the BRAM blocks. The arrows in Figure 3(e) show an example of $Move\_E$ instruction from register $R1$ to $R2$. Considering Figure 3 (b), this instruction should move the data from register $R1$ of $PE1$ to $R2$ of $PE2$, $R1$ of $PE2$ should move to $R2$ of $PE3$ and the same for the rest of PEs. For the $PE1$, $PE5$, $PE9$, and $PE13$ which are on the edge, the west I/O block's data will be moved to $R2$ of these PEs. Also, for the $PE4$, $PE8$, $PE12$, and $PE16$, their $R1$ registers are respectively moved to the $R2$ of $PE1$, $PE5$, $PE9$, and $PE13$ of their right PE-block.

Considering the described high-level abstraction of data movements, the implementation details of the $Move$ instructions are now discussed. As already mentioned, the $NEWS\ Move$ instructions are implemented using bit-serial read/write operations from/into the BRAM blocks. It is shown in Figure 3 (e) that in the design, the registers are stored vertically in the BRAM blocks. This implies that to read a register, the individual bits are read and written one by one. This is because in the standard BRAM read and write operations, data is accessed row by row (not columns) at each clock cycle. Therefore, to read the whole N bits of a register in the vertical implementation, we read the whole row and extract the one bit we need from that row and then repeat the same for the next rows. Therefore, assuming utilizing the registers of size N (bits 0 to $N-1$ in Figure 3(e)), any $Move$ instruction will take $2N$ clock cycles (N clocks for reading and N clocks for writing N bits one by one). We have optimized this operation by utilizing a True Dual-Port BRAM that supports reading and writing two bits at each clock cycle. This speeds up the $Move$ instructions by a factor of $2\times$. In this optimized method, the $Move$ instructions take N clock cycles ($\frac{N}{2}$ clocks for reading and $\frac{N}{2}$ clocks for writing N bits two by two). As mentioned, the PEs of each PE-block share the same BRAM. Therefore, when performing the $Move$ instructions, each PE inside a PE-block can access to the register files of the other PEs inside the same PE-block using direct BRAM reads and writes. However, for the edge PEs of the PE-blocks and tiles, the communications is conducted using data buffers. In this case, the data is read from the adjacent PE-block PEs, stored into the data buffers, and then is written into to the other edge PEs. For the processor arrays edge PEs (the PEs on the first row and first column of the 2-D

processor array), the data comes from the I/O buffers.

3.8 I/O Buffer (Parallel/Serial Converter)

These modules are shown in Figure 2 at the *NEWS* sides of the processor array. The external side of the I/O buffers on the processor array periphery includes serial-to-parallel and parallel-to-serial corner-turn registers to translate between the column-major bit-serial register formats of the edge PEs with external parallel modules such as DRAM and the *Sigmoid* and *Tanh* modules. I/O buffers are provided at the four edges of the processor array to move data into and out of the PEs' register files. Due to the bit-serial nature of the processor array, the I/O buffer provides a serial I/O interface to receive or send concurrent serial data from the edge PEs. In the original design, there was an I/O buffer for the whole edge PEs. Given the high fan-out of such a module, the operating frequency was limited. The design is then optimized so that the I/O buffers are implemented as parallel modules with one associated per each PE-block. This configuration was implemented to enable the processor array to achieve the maximum possible operating frequency. However, this optimized implementation utilizes more resources. The I/O buffers read two serial bits at a time from the edge PEs, store them in a parallel register, read the next two bits of those edge PEs, and store them in the same parallel register until the total N bits of the operands data-width is converted from serial to parallel. The same applies when a parallel to serial conversion is conducted. The I/O buffer modules have a serial and a parallel input and a serial and a parallel output. Assuming data-width of N, the serial input and the serial output size is 2×4 (2 bits at each cycle and 4 PEs on the edge of each PE-block). The parallel input and the parallel output size is $4 \times N$ (4 PEs on the edge of each PE-block and N bits data-width).

3.9 Activation Functions

As shown in Figure 2, the activation functions of *Sigmoid* and *Tanh* are implemented using separate modules outside the processor array. The I/O buffers interface with these parallel activation function modules. The far-right column of PEs feeds the activation functions, and the

results are returned into the top row of PEs. The input to the activation functions comes from the east I/O buffer. The parallel output from the activation functions is converted to serial using the north I/O buffer and then is serially sent to the processor array. Activation functions on FPGAs are commonly implemented using linear approximations of *Taylor* [26], *HARD* [27], and *PLAN* [28] methods. The *Taylor* and *HARD* methods require multiplication and division operations, whereas the *PLAN* method only uses logical shift and addition operations. In our design, for the *Sigmoid* function, the *PLAN* method is implemented since it results in lower execution time and resource requirement than other methods. The *Tanh* function is also implemented based on the *PLAN* method, considering that based on (3), *Tanh* can be computed using *Sigmoid* function [29]. The array of activation functions includes a separate module instance of *Sigmoid* and *Tanh* per each PE-block. Therefore, the outputs from the right-most column of PEs are processed in parallel by the activation functions. For the CNN networks, the *relu* activation function is implemented using the *relu* instruction in the processor array's ISA.

$$Tanh(x) = 1 - 2 \times \sigma(-2x) \tag{3}$$

The *PLAN* method of the *Sigmoid* module, is implemented using an FSM-based method based on (4). The input and output to this module are parallel N-bit width values. In the first state of these FSM, the input range is specified. The next state compute the output value based on (4). In all these conditions, the multiplication is implemented using a shift operation and is added to a constant value. For example, the $0.25 \times x$ operation is a 2-bit shift right, and the $0.125 \times x$ operation is a 3-bit shift right operation. Finally, in the last state, the *ready* signal is set and the controller will notify the north I/O buffer to convert the parallel values to serial and store them in the top row of PEs.

$$\sigma(x) = \begin{cases} 1 & |x| > 5 \\ 0.03125 \times |x| + 0.84375 & 2.375 \leq |x| < 5 \\ 0.125 \times |x| + 0.625 & 1 \leq |x| < 2.375 \\ 0.25 \times |x| + 0.5 & 0 \leq |x| < 1 \end{cases} \tag{4}$$

3.10 Software Programmability

This section elaborates more on how our overlay is software programmable. More specifically, the implementation details of the software macros that are presented in Table 5 are discussed in this section. Figure 6 shows how a simple small CNN network is mapped and executed on the processor array overlay. For more background information on CNN networks refer to [30]. Shown in Figure 6, the input feature maps of three nodes are mapped into the PEs of the processor array. Assuming a 6×6 feature map that produces a 4×4 output feature map ($kernel\_size = 3$, $padding = 0$, $stride = 1$). The three 6×6 feature maps are mapped into 12×12 PEs. The convolution algorithm is implemented using a single multiplication on all PE inputs contained in $R1$ and the weights in $R2$. Partial products are stored in $R3$. In the following step, the partial products are added using $Move\_E$, $Move\_S$ and add instructions with results stored in $R4$, $R5$, and $R6$ for each feature map. This is shown in Figure 6 (a) for only the first feature map. The other two feature maps follow the same operations and the results are saved in $R5$ and $R6$. It should be noted that no additional read and write is required to move the highlighted values into the top-left of the processor array as the final results are mapped and saved in the correct position. Finally, by accumulating the $R4$, $R5$, and $R6$ registers (Figure 6 (b)), the output feature map is stored in $R4$ (Figure 6 (c)). The bias parameters are added in the next step and then the $Relu$ function is applied. The pooling (if any) is then applied to the output feature map (Figure 6 (c)) (not represented in this simple example). In a SIMD architecture, the convolution algorithm is implemented in parallel over the number of nodes for each CNN layer. For larger CNNs, where the feature maps do not fit into the available processor array, multiple registers are used to store

the input feature maps. This reduces the transfer latency swapping feature maps between DRAM and BRAM. For example, assuming feature maps of 6×6 on a processor array of 12×12 PEs, if the number of input nodes is 5, the first 4 feature maps can be stored in $R1$ (same as the prior example) and their associated weights stored in $R2$. However, the last feature map cannot be stored in $R1$ as the array is fully utilized. In this case, the last feature map can be stored in a different register (e.g. $R10$) on the top-left position of the processor array. The convolution algorithm is first applied on registers $R1$ and $R2$ (for nodes $1-4$) with the inputs in $R1$ and their associated weights stored in $R2$. A second convolution can then be applied on $R10$ and $R2$ for the 5_{th} node. In essence, a virtual processor array larger than the physical array can be defined, with each physical PE operating as multiple virtual PEs.

Figure 7 shows the execution of a typical matrix-vector multiplication operation as the main operation of MLP/LSTM/GRU networks within our processor array overlay. For example purposes, we show a small 3×4 matrix W multiplied by vector $X_{4\times1}$ resulting vector $Y_{3\times1}$. Figure 7 (a) shows how the weight matrix W is partitioned into distributed BRAMs of the 2-D array and how the elements of vector X are mapped and replicated into the processor array. One SIMD parallel multiplication is performed (Figure 7 (b)) on all PEs, generating all partial products in one instruction. The addition of partial products is then followed using *Move* and *add* instructions (Figure 7 (b)). For this step, the addition of partial products uses the binary reduction tree shown in Figure 7 (b). The final output from the multiply-accumulate step is then sent to the output buffer (Figure 7 (c)). The partial product accumulation step (Figure 7 (b)) is the most time consuming part in the MVM operation. There is a loop as the number of matrix columns (in this example 4) that performs the *Move_E* and *add* instructions. Using the binary tree reduction method for the addition operations, assuming the matrix size if $n \times m$, the number of *add* instructions is $logm$ and the *Move_E* instructions are as the number of the processor array's column size. In case the matrix column size if not a power of two number, the closest power of two to the matrix column size is found and then the binary tree reduction step is performed on those number of columns. The rest of columns are processed using linear reduction method. For

example, if the column size is 25, 16 of them are processed using binary tree reduction method and the rest 9 columns are processed using linear reduction method. It also should be noted that the in case the matrix column number is less than the processor array size, the final result would be in one of the middle columns of the processor array and it should be moved to the last columns of PEs using a series of $Move\_E$ instructions. This operation is further optimized for internal data movement to support a binary tree interconnect for the $Move\_E$ instructions as well as the *add* instructions. This is disscussed in more detail in Section 5.1.

The larger matrix multiplications that could not fit into the 2-D processor array are implemented using partitioning (divide-and-conquer) so that each subsection is stored in a different register. After the matrix multiplication for each subsection, the results are merged together to form the final output. In this case, the matrix size is larger than the processor array dimensions and needs to be divided into smaller sub-matrices that could fit into the processor array. A simple example of this case is shown in Figure 8. In this example, assume that a matrix of size 4×6 is multiplied by a vector of size 6×1 on a processor array of size 2×3. Therefore, since the matrix and the vector do not fit into the processor array, they are divided into smaller 2×3 and 3×1 sub-matrices. Each sub-matrix is stored in a different register. The normal MVM operation is performed on the small sections and then the results are added to generate the final output 4×1 vector. This vector is also stored in sub-vectors of size 2×1, each in a different register. Vector-vector element-wise operations (*VVA* and *VVM*) are performed using a single SIMD instruction on the whole vector. In case of large vectors, the divide-and-conquer method is applied and the vector is divided into smaller sub-vectors, each is stored in a different register. In this case, for example for the *VVA* operation, the number of additions is one per sub-vector. The number of instructions for each software macro is presented in Table 21.

There are some software macros that move the data between different edges of the 2-D processor array. They include *ColumntoRow* for moving data form the last column of PEs to the first row of PEs, *RowtoColumn* for moving data from the first row of PEs to the last column of PEs, and *ColumntoColumn* for moving data from the last column of PEs to the first column of

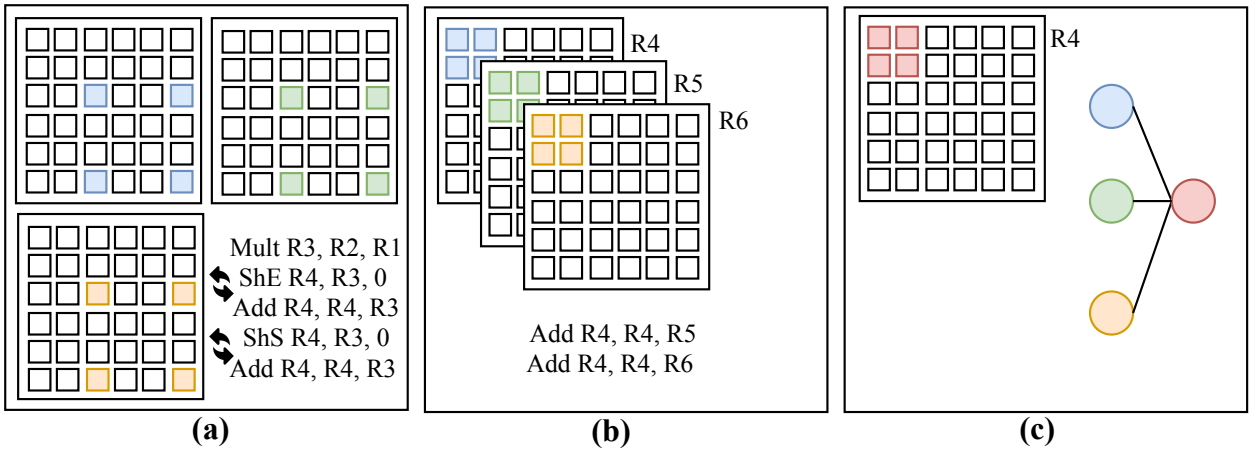

Figure 6: Mapping convolution algorithm into processor array.

PEs. These software macros replace a series of *NEWS Moves* instructions with a single instruction. They are utilized for implementing the ML algorithms when the output of a layer needs to be used in the next layer. These software macros are also used when applying the activation functions. There is an input variable for these software macros that determines if the data needs to go through the activation functions and then be stored into the PEs register files or the data should be directly moved between the PEs register files without going through the activation functions.

The other set of software macros are used to store data into the right location of the PEs register files. For example, in Figure 7, the elements of matrix W and vector X are written into the register $R1$ and $R2$ of the PEs in a proper way. The input to this software macro is the matrix/vector, the destination register name, and the method of storing that matrix/vector. In this example, the elements of matrix W are stored as a 2-D array, while the elements of vector X are first stored in the first row of PEs and then are copied to the below PEs. Vector X is copied since it is should be multiplied by all rows of matrix W to be able to perform the WX matrix-vector multiplication operation. The other method of storing a vector is for example storing it in the last column of PEs. This is mostly used for storing the biases values. Therefore, this software macro that is called $Write\_Matrix$, is used to store the network parameters into the PEs register files. It is called before running the other software macros.

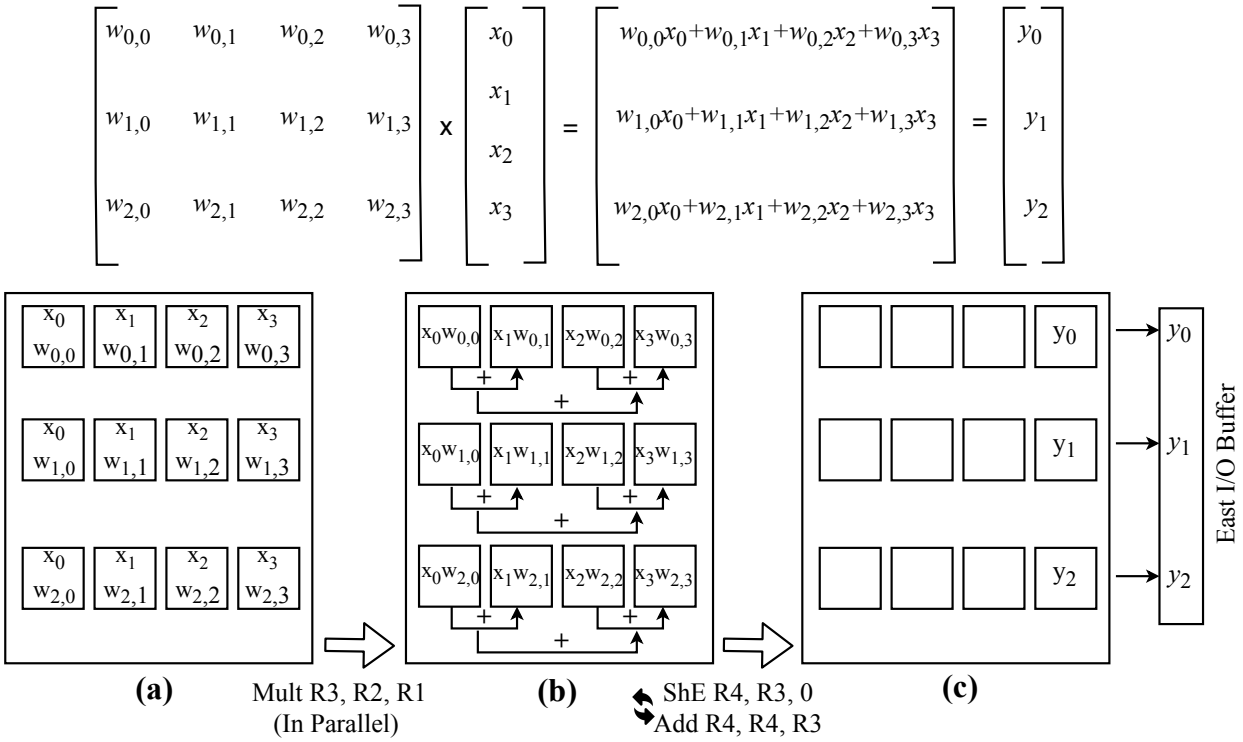

Figure 7: Mapping matrix multiplication algorithm into processor array.

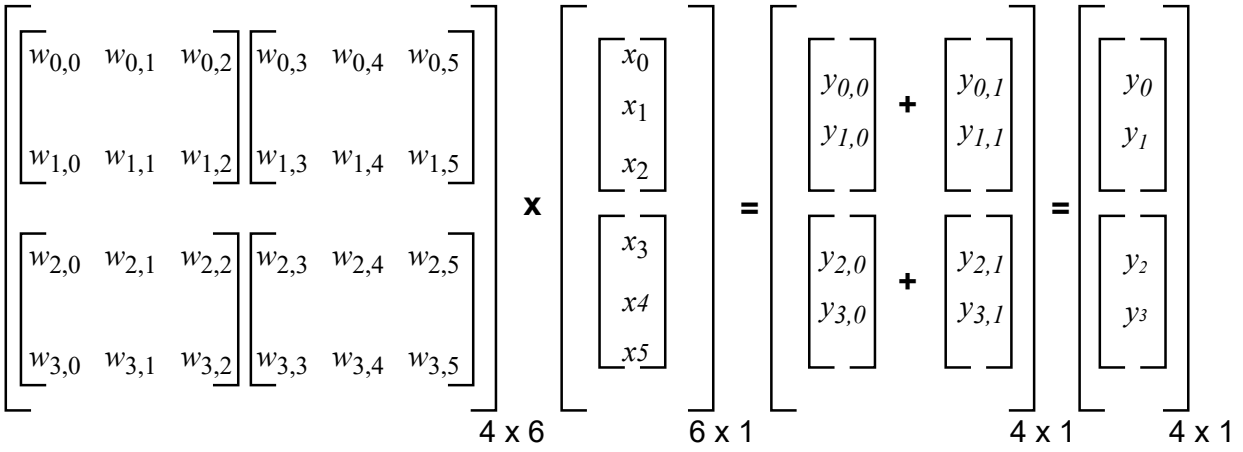

Figure 8: Mapping large matrix multiplication algorithm into processor array.

Table 5: ISA and Software Macros

ISA Description	
Instruction Format	**Description**
$add\,R_d, R_{s_1}, R_{s_2}$	$R_d = R_{s_1} + R_{s_2}$
$sub\,R_d, R_{s_1}, R_{s_2}$	$R_d = R_{s_1} - R_{s_2}$
$mult\,R_d, R_{s_1}, R_{s_2}$	$R_d = R_{s_1} \times R_{s_2}$
$Move\_E\,R_d, R_{s_1}, 0$	moves all PEs' R_{s_1} to R_d of their right PE
$Move\_W\,R_d, R_{s_1}, 0$	moves all PEs' R_{s_1} to R_d of their left PE
$Move\_N\,R_d, R_{s_1}, 0$	moves all PEs' R_{s_1} to R_d of their above PE
$Move\_S\,R_d, R_{s_1}, 0$	moves all PEs' R_{s_1} to R_d of their below PE
$relu\,R_d, R_{s_1}, 0$	$R_d = R_{s_1}\, if\, R_{s_1} > 0\ else\ R_d = 0$
Software Macros	
Function Name	**Description**
$MVM(R_d, R_{s_1}, R_{s_2})$	Matrix-Vector Multiplication
$VVA(R_d, R_{s_1}, R_{s_2})$	Vector-Vector Element-wise Addition
$VVM(R_d, R_{s_1}, R_{s_2})$	Vector-Vector Element-wise Multiplication
$2D\_conv(R_d, R_{s_1}, R_{s_2})$	2-D Convolution (any kernel and stride size)
$2D\_pad(R_d, R_{s_1}, 0)$	2-D padding (any padding size)
$2D\_max\_pool(R_d, R_{s_1}, 0)$	2-D max pooling (any kernel and stride size)
$2D\_avg\_pool(R_d, R_{s_1}, 0)$	2-D average pooling (any kernel and stride size)

Table 6: Software Macros Instruction Count Break-Down.

Software Macro*	**Instruction**		
	add	*mult*	*Move*
$MVM_{(m_{i+1} \times m_i)}$	$\log_2 m_i$	1	x
$VVA_{(m_i)}$	1	0	0
$VVM_{(m_i)}$	0	1	0
$2D\_conv_{(m_{i+1} \times m_i)}$	$m_i + m_{i+1} - 2$	1	$m_i + m_{i+1} - 2$
$2D\_max\_pool_{(m_{i+1} \times m_i)}$	0	0	$m_i + m_{i+1} - 2$
$2D\_avg\_pool_{(m_{i+1} \times m_i)}$	$m_i + m_{i+1} - 2$	0	$m_i + m_{i+1} - 2$
$ColumnToRow$	0	0	2
$RowToColumn$	0	0	2
$ColumnToColumn$	0	0	2

* Numbers in parenthesis are the matrix/vector sizes.

Chapter 4

Results

In this section, we evaluated the processor array overlay from different aspects. All experiments used the Xilinx Vivado 2018.3. We first started with evaluating the functionality and the performance of the design by running some ML benchmarks. The results for these several standard ML benchmarks are presented and compared with the previous equivalent custom point designs. The training was conducted on Tensorflow, and the computed network parameters (weights and biases) were exported and used in the inference phase. The C code versions of the benchmarks were developed for the inference phase of each ML network and were run through the MicroBlaze compiler to generate assembly instructions for the developed system. The reported performance results of the inference phase were run-time latencies measured using accurate cycle counters running at 130 MHz implemented on a Virtex-7 VC707 (xc7vx485T-2ffg1761) as well as running at 200 MHz operating frequency on a Virtex Ultra FPGA (xcvu9p-flgb2104-2-i).

The benchmarks that are explored include two LSTM networks with different sizes, an MLP, a CNN network (SqueezeNet v1.1), and a GRU network. The [31] is an LSTM network with input size of 61, three hidden layers of size 250, and output size of 39 used for speech recognition on TIMIT dataset [32]. The [34] benchmark is another LSTM network with an input layer of size 64, two hidden layers of size 128, and the output size of 64. This network is used for character recognition on the Shakespeare dataset [35]. The MLP [37] includes 784 inputs, hidden layer and output sizes of 100 for image recognition on MNIST dataset [38]. The SqueezeNet v1.1 CNN network [39] is used for image classification on ImageNet dataset [40] and the GRU network [42] with 39 inputs, two hidden layers of sizes 256 and 200, and 10 outputs is also utilized in speech recognition on [43] dataset. The benchmarks are compared to the custom FPGA-based accelerations to evaluate how the overlay method works in terms of resource utilization and performance that have been considered as inherent problems of overlays. The results show that our method provides lower inference latency and lower resource utilization than most

benchmarks.

A summary of results in terms of performance and resource utilization (LUTs, FFs, BRAMs, and DSPs) comparing our method to other custom designs provided in Table 7. The developed design can implement each ML benchmark in different data-widths, but only the results for the data-widths reported in comparative works are included. The processor array of up to 16k PEs have been implemented on the Virtex-7. However, the resource utilization in Table 7 is for an array of 10k PEs. This size of the processor array was chosen to match the largest benchmark reported in the literature. A smaller processor array could have been used for some of the benchmarks, which would result in better resource utilization numbers. The choice of using a single-size processor array was made in part to show how the overlay can implement all benchmarks on the same design by rewriting software instead of resynthesizing.

4.1 Latency Comparison

The results in Table 7 shows that the overlay design achieved a 34.2× speedup in LSTM(1) compared to the HLS-based design reported in [33]. This speedup is achieved at a lower clock frequency (130 MHz versus 150 MHz). LSTM(2) shows that our design achieved a 3.5× speedup compared to the HDL-based design reported in [36]. The ability to decrease execution time using bit-serial arithmetic points out the importance of the data movement overhead. This is also shown when comparing the execution time improvement of our method to LSTM(1) and LSTM(2). In case of LSTM(2) since all network parameters could fit into the on-chip BRAM memory of the FPGA, less improvement is achieved than LSTM(1) in which some parameters are stored in DRAM. Therefore, the decrease in the overall execution time of the developed method is attributed to the memory access times reduction. The design for the CNN network achieved a modest 2.1× speedup compared to results reported in [41]. Execution time results were not reported in [20]. The achieved modest 2.1× speedup also confirms that since the CNN benchmarks are computation-bound, the design's speedup is less than the communication-bound ML networks when compared with the previous works. Our design achieved a speedup of 4.3×

for the MLP network and a 12.5× speedup compared to the results reported in [44] for the GRU network. The clock frequencies were approximately equal (130 MHz for our design versus 125 MHz in [44]). Different latencies were reported in [44] for various utilized delta thresholds. The reported comparisons are against a delta threshold of $0x00$, which is equivalent to what is implemented in our design. In summary, the achievable speedups for MLP/LSTM/GRU networks are higher compared to CNN networks. This is a result of the design SIMD ability to exploit parallelism, especially in MVM operations. In MLP/LSTM/GRU networks, the multiply step in a complete MVM operation occurs once. However, the parallelism within a CNN network is dependent on the number of nodes in a given layer, with each layer being computed sequentially. Moreover, in CNNs, feature maps are added together in each stage.

Table 7: FPGA Implementation Results

Name	Exe Time	Data * Format	LUTs	FFs	BRAMs	DSPs	Freq (MHz)	FPGA	Method
LSTM(1) (61, 250, 250, 250, 39) [31] on TIMIT dataset [32]									
This work	17.5 ms	FxP 32	138380	67801	313	0	130	Virtex-7	Overlay
This work	11.4 ms	FxP 32	133890	56207	313	0	200	Virtex Ultra	Overlay
[33]	390 ms	FlP 32	198280	182646	1072	1176	150	Virtex-7	HLS
LSTM(2) (64, 128, 128, 64) [34] on CharRec dataset [35]									
This work	395.9 us	FxP 16	138380	67801	313	0	130	Virtex-7	Overlay
This work	257.1 us	FxP 16	133890	56207	313	0	200	Virtex Ultra	Overlay
[36]	900 us	FxP 16	7201	12960	16	50	142	Zynq	HDL
MLP (784, 100, 100) [37] on MNIST dataset [38]									
This work	0.5 ms	FxP 32	138380	67801	313	0	130	Virtex-7	Overlay
This work	0.3 ms	FxP 32	133890	56207	313	0	200	Virtex Ultra	Overlay
[37]	1.3 ms	FxP 25	139562	175604	50	400	100	Virtex-7	HDL
CNN SqueezeNet v1.1 [39] on ImageNet dataset [40]									
This work	51.0 ms	FxP 8	138380	67801	313	0	130	Virtex-7	Overlay
This work	33.1 ms	FxP 8	133890	56207	313	0	200	Virtex Ultra	Overlay

[41]	70.5 ms	FxP 8	34489	25036	97.5	172	100	Zynq	HLS
[20]		FxP 8	173522	241175	193.5	704	200	Kintex-7	Overlay
GRU (39, 256, 200, 10) [42] on DeepSpeech dataset [43]									
This work	3.3 ms	FxP 16	138380	67801	313	0	130	Virtex-7	Overlay
This work	2.1 ms	FxP 16	133890	56207	313	0	200	Virtex Ultra	Overlay
[44]	26.4 ms	FxP 16	261357	119260	768	457.5	125	Zynq-700	HDL

* FxP means fixed-point and FlP means floating-point.

Table 8: Low-precision Networks Performance Comparison.

Name	Network Design	Latency (us)	Data Format	Freq. (MHz)	FPGA	Method
Xilinx FINN [46]	MLP (1024)	2.44	FxP 1	200	Zynq-7000	HDL
This work	MLP (1024)	6.60	FxP 1	200	Virtex Ultra	Overlay
Xilinx FINN [46]	MLP (256)	0.31	FxP 1	200	Zynq-7000	HDL
This work	MLP (256)	0.45	FxP 1	200	Virtex Ultra	Overlay
Intel [1]	LSTM (512)	1.16	FxP 8	275	Stratix-10	HDL
Microsoft BW [**?**]	LSTM (512)	3.08	FxP 8	200	Stratix-10	Overlay
This work	LSTM (512)	5.50	FxP 8	200	Virtex Ultra	Overlay
[47]	MLP (500)	27.77	FxP 2	200	Stratix-10	HDL
This work	MLP (500)	37.18	FxP 2	200	Virtex Ultra	Overlay

To further evaluate our design in terms of latency, the overlay is programmed to run some low-precision networks. The benchmarks include MLP and LSTM networks running at 1-8 bit data-width. As shown in Table 8, although the design's latency is more than the other works, our method provides the flexibility and the reconfigurability that enables running all benchmarks without the need to re-synthesize. Moreover, the latency is in the same order of magnitude as these industry works. The reason of higher latency in our design is using bit-serial operations. For instance, in the bit-parallel implementation at 8-bit data width, a single addition only takes 1 clock cycles. However, in our overlay a single addition instruction takes 8 clock cycles. This increased instruction latency has to some extent compensated with our SIMD-based architecture. However, for these large benchmarks, the network is required to be divided into smaller sub-sections and therefore some serial operations are performed for the MVM operations. Moreover, some of these previous works are customized designs as opposed to our overlay design.

4.2 Resource Utilization Comparison

The resource utilization of the developed overlay with $10k$ PEs on the Virtex-7 FPGA is broken down into more detail in Table 9. This table shows that in our overlay, a single bit-serial ALU only takes 2 LUTs and 2 FFs to be implemented. As a result, a single PE-block and a tile would

be small to allow more PEs be packed into the FPGA. The number of each module in the design is also reported in this table. The 100×100 design contains 5×5 tiles, with each tile including 5×5 PE-blocks. Each PE-block includes 16 PEs (4×4) which results in $5 \times 5 \times 4 = 100$ PEs for each side of the processor array and a total of $100 \times 100 = 10k$ total PEs. There are 16 ALUs per PE-block for a total of $16 \times 25 \times 25 = 10,000$ (25 tiles, each includes 25 PE-blocks) ALUs for the complete design. The processor array includes a parallel-to-serial I/O buffer (P/S converter) per PE-block on each side of the 2-D array which results in a total of $25 \times 4 = 100$ P/S converters. The number of activation functions is a module for each PE and there is 100 modules on only the east side of the 2-D array. It also should be noted that the summation of the provided resource utilization per module does not result in the total resource reported for the whole design as some modules are included and counted inside others. If added together they will be counted twice or more. BRAM resources are only utilized within the PE-blocks to present the register files. Each PE-block has one BRAM with a size of 16×1024 bits. Multiplying the total PE-blocks by the number of BRAMs per PE-block results in $1 \times 625 = 625$ BRAMs. The reported number of BRAMs in Table 9 is based on BRAM18E1 while Table 7 is based on BRAM32E1 of a Virtex-7 FPGA. Table 9 also shows that no DSPs are used in the processor array while as shown in Table 7, the previously proposed custom designs use DSPs. This is a significant difference between our design and the HLS driven designs. While DSPs provide reduced latency for full-precision operations, they can limit concurrency and result in inefficient resource utilization for less than full-precision operations. Experiments on a Virtex-7 FPGA show that utilizing DSPs for the arithmetic operations limits the number of PEs to 676 in a 32-bit full-precision ALUs, whereas using LUT-based bit-serial ALUs results in up to $16k$ PEs on the same FPGA.

4.3 Performance Comparison

For completeness, we include Table 10 which provides additional information on the latency and performance of different arithmetic instructions for bit-widths ranging from 32 to 8 bits on a processor array of size $10k$. The ALU's performance is reported using

Table 9: Our Processor Array Overlay's Resource Utilization

Module	LUTs	FFs	DSPs	BRAMs	# Modules
ALU	2	2	0	0	10,000
Controller	967	144	0	0	26
PE-block	11	64	0	1	625
Tile	3839	1833	0	0	25
P/S Converter	900	175	0	0	100
Sigmoid	18	32	0	0	100
Tanh	195	157	0	0	100
SPAR-2	138380	67801	0	625	1

Giga-Operation-Per-Second (*GOPS*) metric. In our design, the values of the *GOPS* metric depend on a couple of factors including the operating frequency, data-width, the instructions latency as well as the total number of PEs in the design. Therefore, this metric evaluates the trade-off between the mentioned factors. Shown in Table 10, the add/sub and $Move$ operations take fewer clock cycles than *mult* instruction and therefore resulting in higher *GOPS*. The radix-4 Booth multiplication has a higher *GOPS* and lower clock cycles than the radix-2 method as the number of iterations in this method is half of the radix-2 method. The linear relation between the operating frequency (130 MHz versus 200 MHz) and the processor array's *GOPS* is also shown in Table 10. Moreover, there is also a linear relation between the data-width and the clock cycles spent for add/sub and $Move$ instructions. However, for the *mult* instruction, there is an exponential relation between the data-width and the number of clock cycles. For every instruction, the *GOPS* values decrease as the data-width increases. Because the instruction's execution time increases with the increased data-width. The effects of number of PEs on the processor arrays's *GOPS* performance is also presented in Table 10 for 16k versus 65k PEs. Comparing the *GOPS* values for a specific on the same operating frequency and different number of PEs show that there is linear relation between the number of PEs and the *GOPS* values. For example, at 8-bit data width and 130 MHz, when increasing the number of PEs from 16k to 65k (4$\times$ increment), the *GOPS* values are increased by around 4$\times$ as well from 132.98 to 531.94 for the add/sub instruction, or from 26.59 to 106.38 for the *mult* instruction.

Table 10: Bit-serial ALUs Performance

	Clocks (#)	**GOPS (16k PEs)**		**GOPS (65k PEs)**	
		130 MHz	200 MHz	130 MHz	200 MHz
8-bit Data-Width					
Add/Sub	16	132.98	204.80	531.94	819.20
Mult radix-2	144	14.77	22.75	59.10	91.02
Mult radix-4	80	26.59	40.96	106.38	163.84
Move	8	265.97	409.60	1063.89	1638.40
16-bit Data-Width					
Add/Sub	32	66.49	102.40	265.97	409.60
Mult radix-2	455	3.91	6.02	15.64	24.09
Mult radix-4	288	7.38	11.37	29.55	45.51
Move	16	132.98	204.80	531.94	819.20
32-bit Data-Width					
Add/Sub	64	33.24	51.20	132.98	204.80
Mult radix-2	2112	1.00	1.55	4.02	6.20
Mult radix-4	1088	1.95	3.01	7.82	12.04
Move	32	66.49	102.4	265.97	409.60

4.4 Overlay Portability

Overlays such as our design bring programmability and portability into an FPGA design. Table 7 presents the result of implementing our design on a Virtex Ultra FPGA. Once the overlay was synthesized on the Virtex-7, the same code was used to run without modification on the Virtex Ultra. As the Virtex Ultra is larger than Virtex-7, more PEs ($65k$) can be fit into the chip. However, to show portability, the same array of $10k$ PEs used on the Virtex-7 was also used on the Virtex Ultra. Not unexpected, synthesizing the array on the Virtex Ultra yielded a higher clock frequency (130 MHz on the Virtex-7 versus 200 MHz on the Virtex Ultra). The limitation on the operating frequency of our design to 130 MHz on the Virtex-7 was due to BRAM placement and routing. The critical path of the design was the connections between the PEs and BRAMs. These connections produced high congestion on the Virtex-7 board which is resolved on the Virtex Ultra without any changes to the design logic. The higher clock frequency achieved on the Virtex Ultra translated into higher throughput of the bit-serial arithmetic operations. In addition to running our design on Virtex Ultra, it is also ran on the Zynq Ultrascale+ (ZCU104) FPGA to further evaluate

the design portability between different families of the FPGAs and using different soft processors. On a Zynq FPGA, the overlay is ran on a ARM processor instead of the MicroBlaze. The design is functionally validated, but the results are not included since this FPGA has a lower number of LUTs and results in smaller processor arrays than on the Virtex-7 and Virtex Ultra FPGAs.

Chapter 5

Optimizations

5.1 Internal Data Movement

In ML inference acceleration, the computation/communication ratio determines the end-to-end inference latency seen by a user. In this section, we discussed on how our processor array was optimized for data communication to reduce the inference latency. Table 11 breaks down the percentage of cycles spent in computations (Array Active Cycles) and communications (Internal Data Movement and Weight Stall Cycles) within our overlay. In this table, the Array Active Cycles include the arithmetic operations (MAC operations), Internal Data Movement represents the communication latency for moving the data between the PEs, and the Weight Stall Cycles is the DRAM access time. Based on Table 11 and consistent with results reported in [24], the MLP/LSTM/GRU benchmarks are communication-bound. In these benchmarks, the percentage spent on Internal Data Movement and Weight Stall Cycles is higher than the Array Active Cycles. This is opposed to CNN networks that most of the total latency is spent on the Array Active Cycles. Without any loss to the generality, a system designer targeting such communication-bound networks may want to perform some additional domain customizations to further reduce inference latency for their applications. Amdahl's law would point in the direction of the communications subsystem. In our processor array, the standard $NEWS$ communication network is general but requires multiple shifts in the width and height dimensions to move data between the PEs. These customizations can be encapsulated in software macros as part of a domain-specific library available to programmers. The CNN network is not included as the Internal Data Movement is a small portion of its total execution cycles. The binary tree reduction network is implemented using data movement between BRAM blocks. There is no physical connection between the processing elements. Figure 9 shows how the binary tree network connects the PEs. In the original design with standard $NEWS$ interconnect network, the data can move to the adjacent PEs that are only 1 PE away from the source PE. Therefore, to move data to

further PEs, the data needs to go one-by-one from the source to the destination PE using a series of *Move* instructions. In the optimized PE interconnection network, the data can be moved from any PE to the PEs that are 2, 4, 8, or any other distance which is a power of two, in the *NEWS* directions. Using this method, when moving data from further PEs, there is no need to hop through all the middle PEs. This will reduce the execution time since a sequence of *Move* instructions can be replaced by a single *Move* instruction. In this optimization, the number of *Moves* is set in a register which is a power of two.

To implement this optimization in our memory-centric design, the data needs to be moved between the BRAM blocks (register files) of different PE-blocks. Previously, with the original design, since the data movement was conducted on the PEs that are only 1 hop away, the data movement was performed inside the PE-blocks in one BRAM block not between different PE-blocks. In the optimized method, if the requested PEs distance is 1 or 2, the data movement is still performed inside each PE-block. Since the PE-blocks are 4×4, the PE with distance 1 and 2 are inside the same PE-block. If the requested PEs distance is 4, for each destination PE, the data comes from the PE in the same position of the destination PE, but in the adjacent PE-block. For example, when running the (*Move_E R*2, *R*1, 4) instruction, the data should be moved to the PEs that are in the distance of 4 in the east direction. In this case, the data from PE1 of a PE-block (Figure 3) should be moved to PE1 of the PE-block on the east side of the current PE-block. If the requested PEs distance is 8, 16, or 32, for each destination PE, the data comes from the PE in the same position of the destination PE, but in the PE-block that are 2, 4, or 8 PE-blocks further from the source PE. The same pattern applies to the rest of PEs destination values. In these case, the destination PE-block's location is computed by dividing the destination value by 4. After determining the destination PE-block's location, the data is read form that PE-block, stored in a buffer, is sent to the destination PE-block as an input port, and stored in the correct location in the BRAM's register file of the destination PE-block. The effects of implementing this optimization on the latency and resource utilization of our overlay is discussed in the following sections.

Table 11: Breakdown of Execution Cycles

Operation	LSTM(1)	LSTM(2)	MLP	CNN	GRU
Array Active Cycles[a]	33.6%	23.9%	54.2%	84.1%	10.8%
Internal Data Movement[b]	37.8%	76.1%	45.8%	0.1%	27.6%
Weight Stall Cycles	28.6%	0%[c]	0%[c]	15.8%	61.6%

[a] Multiply-Accumulate (MAC) operations.
[b] *NEWS* operations.
[c] No data movement from DRAM to BRAM.

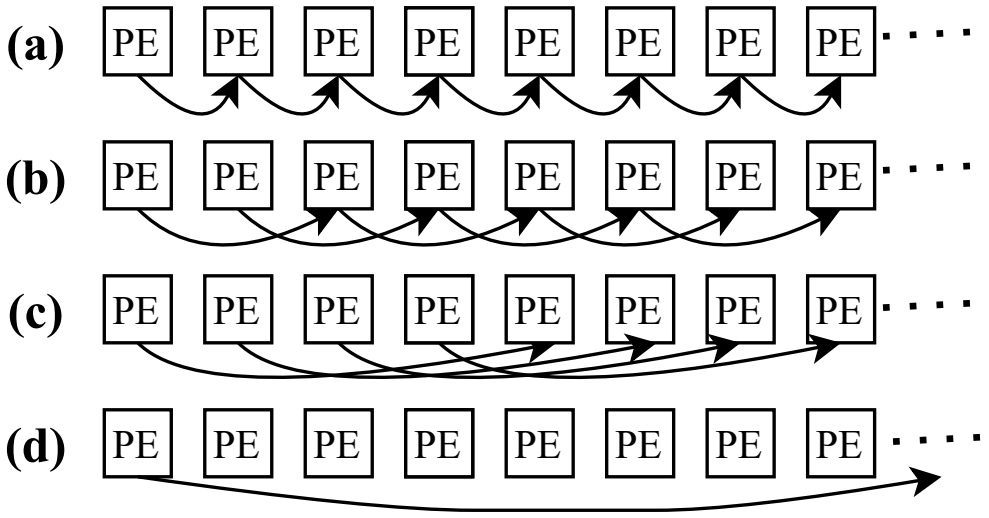

Figure 9: Binary Tree Interconnect.

5.1.1 Latency Comparison

The comparison of utilizing binary tree interconnection as opposed to linear interconnection between the PEs is shown in Figure 10. The total number of clock cycles for each of the same benchmarks are compared to show the difference between these two interconnections on the Array Active Cycles (MAC operations) and Internal Data Movement (*Move*) operations. *MAC* includes the number of multiply and accumulations (*add* and *mult* instructions) and *Move* represents the total *NEWS Moves* instructions. The blue and the orange diagrams show the number of clock cycles spent on the MAC and Move operations when all accumulations and data movements are implemented using traditional linear operations. The gray and the yellow diagrams represent the number of clock cycles spent on the *MAC* and *Move* operations when both accumulations and internal data movements are performed using binary tree method. Comparing the blue and the gray diagrams in all benchmarks show that, as expected, the number of cycles is decreased because of the lower number of performed additions as the result of utilizing binary tree accumulation method. The same applies to the orange and yellow diagrams that the total clock cycles is reduced as the result of utilizing binary tree interconnect between the PEs.

To evaluate the effects of implementing binary tree optimization on the instructions count, Table 12 shows the number of performed instructions for each benchmark when using the linear or binary tree methods. The *add* and *Move_E* instructions are affected by this optimization since they are used in the matrix-vector multiplication operations. The other instructions remain the same since they have only performed on PEs with distance of 1. The amount of reduction in the number of instructions depends on the benchmark size and if the number of neurons are a power of two or not.

An evaluation on the effects of using binary tree interconnection network as opposed to using larger processor arrays is performed using two different processor array sizes. This evaluation clarifies that is it better to dedicate the available FPGA resources to the PEs or to the PE's interconnections. In these two cases, a smaller processor array is optimized with binary interconnection network, but the larger processor array supports a higher level of concurrency

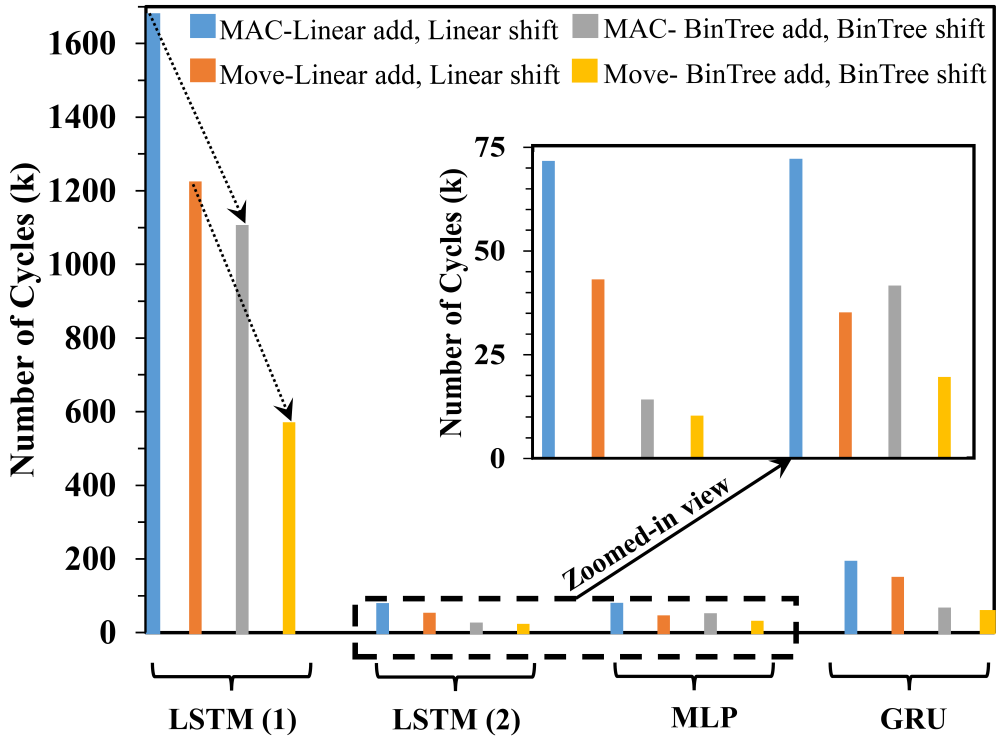

Figure 10: MAC and Move clock cycles at Internal Data Movement optimization.

Table 12: Effects of Internal Data Movement Optimization on Benchmarks Instruction Count.

Instr.	**LSTM(1)**	**LSTM(2)**	**MLP**	**GRU**
(Linear add, Linear shift) / (BinTree add, BinTree shift)				
Move_E	**36259 / 15819**	**2048 / 400**	**889 / 441**	**16304 / 5072**
Move_W	0 / 0	0 / 0	0 / 0	0 / 0
Move_N	30 / 30	2 / 2	0 / 0	16 / 16
Move_S	1544 / 1544	274 / 274	100 / 100	996 / 996
add	**18850 / 10090**	**1657 / 265**	**892 / 452**	**9092 / 3108**
mult	410 / 410	19 / 19	8 / 8	188 / 188

Table 13: Analysis of PEs and Interconnections

MLP Size	Linear Interconnect 200×200 PEs	Binary Tree Interconnect 100×100 PEs
	Latency (μs)	
200	91.36	46.56
700	354.56	120.16
1100	264.32	174.72
1300	385.12	210.88
2200	485.12	331.36
3200	733.44	619.36

with having more PEs. Table 13 shows this comparison for a set of MLP benchmarks running at the same operating frequency of 200 MHz and the same data-width of 32-bit fixed-point. As shown in this table, the latency of the processor array with lower number of PEs ($\frac{1}{4}$ PEs) is less than the processor array with more PEs that is not optimized for internal data movements. These example benchmarks present the importance of the internal data movement optimization for these MLP networks. In the processor array with 100×100 PEs, none of the utilized MLPs fits into the processor array. Therefore, the divide-and-conquer method is used and the matrices and vectors are divided into smaller sections. Therefore, the larger the MLP size is, the more subsections are required. In this case, the subsections are 100×100. For example, the MLP with size 200 is divided into two subsections, the MLP with size 3200 is divided into 32 subsections of size 100×100. This divide-and-conquer method increases the latency. However, since this processor array (100×100 PEs) is optimized for internal data movement, the increased latency as a result of more subsections, is compensated with lower latency for internal data movement operations. It should also be noted that in the larger processor array with 200×200 PEs, the MLPs are still required to be divided into smaller subsections of size 200×200. For example, the MLP with size 200 is not needed to be divided into subsections, the MLP with size 3200 is divided into 16 subsections of size 200×200. In this larger processor array, the number of subsections are less than the other processor array, but since it is using linear interconnects, the MLP benchmarks latency is higher than the other processor array.

Table 14: Effects of Binary Tree Interconnect

Benchmark	Binary Tree add Linear Move	Binary Tree add Binary Tree Move
Execution Time		
LSTM(1) (ms)	11.4	8.2
LSTM(2) (us)	257.1	123.8
MLP (ms)	0.3	0.2
GRU (ms)	2.1	1.2
Resource Utilization (10k PEs)		
LUTs	133890	492937
FFs	56207	76501
BRAMs	313	313
DSPs	0	0

5.1.2 Resource Utilization Comparison

Table 14 shows how the binary tree interconnection, takes additional resources to be implemented on a Virtex Ultra FPGA and how the inference latency of the benchmarks can be reduced by augmenting the *NEWS* network with a binary tree interconnect network. These customizations can be encapsulated in software macros as part of a domain-specific library available to programmers. The CNN network is not included in Table 14 as the Internal Data Movement is a small portion of its total execution cycles. In this table, the LUTs and FFs are increased by 3.9× and 1.3× compared to the original design. The BRAMs and DSPs resource utilization is not changed since the interconnections are implemented using LUTs and FFs. Comparing the LUTs' resource utilization in this table shows that to be to implement the binary tree interconnection with a fixed number of LUTs on an FPGA, the number of PEs would be decreased by 4×. This is also conformed in Table 13 that the optimized processor array is 4× smaller than the original processor array implementation.

5.2 Bit-Sliced Method

As discussed in Section 3.4, the ALUs in the original design were developed using the bit-serial method. Although using bit-serial will increase the level of parallelism in our SIMD design by

Table 15: Instructions Latency in Clock Cycles.

	Addition/ Subtraction	Multiplication	Move ($NEWS$)
1-bit Serial	$2N$	N^2+2N	N
p-bit Slice	$3\frac{N}{p}$	$2(\frac{N}{p})^2+\frac{N}{p}$	$\frac{N}{p}$
32-bit Parallel	3	3	2

packing more PEs into a specific FPGA, the increased number of cycles spent for each instruction in the bit-serial method, inversely affects the total inference latency. To evaluate this design trade-off, we have explored using bit-sliced and bit-parallel operations within the ALUs. In these methods, the ALUs' bit-width would be more than 1 bit. The slice width in our design varies from 2 to 32 bits. In the bit-sliced methods, each slice is processed at a time, and they are connected together serially. The bit-sliced methods spend less clock cycles per instruction since instead of processing 1-bit at a time, they work on p-bit slices. Now, assuming a data-width of N, in the bit-serial method, we perform N iterations on the operand, while in the bit-sliced methods, $\frac{N}{p}$ iterations are performed. The ALU's bit-width should be adapted to the slice size, as they would process one slice with various sizes from 2 to 32 bits.

The execution time of instructions based on the slice's size is shown in Table 15. This table shows that in the bit-serial method, the instructions' latency only depends on the operands data-width (N) that determines the number of iterations. In the bit-sliced methods, the instructions latency depends on the data-width (N) and the slice size (p). The larger the slice size is, the less iterations are required and therefore the instruction's latency is decreased. In the bit-parallel method, the instruction's latency is a fixed number because all bits are processed in one iteration.

5.2.1 Bit-sliced PE-blocks

When using the bit-sliced methods, the layout of each PE-block can vary based on the slice size. Figure 11 shows the PE-blocks' layout for each slice size. In bit-sliced methods, the number of PEs per PE-block depends on the slice size. When using a $RAMB18E1$ BRAM block as the register file, there are 16 columns that can be accessed in parallel. For the bit-serial method, these

16 columns are dedicated to 16 PEs with 1 column per PE. However, when using bit-sliced methods, these 16 columns can be divided into some multi-bit slices. In this case, if the slice size is p, the number of PEs that can be accessed in parallel would be $\frac{16}{p}$ (p columns per PE). Therefore, for example, for slice sizes of 2, 4, 8, and 16, the number of PEs in a PE-block would be $\frac{16}{2} = 8$, $\frac{16}{4} = 4$, $\frac{16}{8} = 2$, and $\frac{16}{16} = 1$, respectively. Different methods on how to layout these number of PEs in a PE-block is presented in Figure 11. As shows in this figure, in 2-bit slices, 8 PEs can be layout in a 4×2 or 2×4 layout. Either of these two architectures could be used. In 4-bit slices, 4 PEs can be layout in a 4×1, a 2×2 or a 1×4 layout. Based on what is discussed about the preference of using square shape PE-blocks, the layout of size 2×2 is used for 4-bit slices. In 8-bit slices, 2 PEs can be layout in a 2×1 or 1×2 layout. Either of these two architectures could be used. In 16-bit and 32-bit slices, 1 PE is used per PE-block and therefore there is only one 1×1 PE-block layout. The PE-blocks are then replicated to form larger processor arrays. The final size of the 2-D processor array for each of the bit-sliced methods depends on the PE-blocks layout. For example, when using 2-bit slices, the PE-blocks layout is 4×2 PEs. Therefore, by replicating the PE-blocks in a 2-D array of 3×3 PE-blocks, results in a 12×6 arrays of PEs. In this processor array, there is 3 PE-blocks on each rwo and each column. Therefore, considering the PE-block's size of 4×2, each row includes $3 \times 4 = 12$ PEs and each column includes $3 \times 2 = 6$ PEs that results in a 2-D array of 12×6 total PEs. The final processor array's layout depends on the PE-blocks' layout. If each PE-block is an square, the processor array would be an square as well.

5.2.2 Bit-sliced Arithmetic

In the bit-sliced methods, the arithmetic operations are performed on one slice at a time. The operations on one slice is performed in parallel, and the connections between the slices are serial. For performing the addition and subtraction instructions in the bit-sliced methods, a slice of each input operand would be the input to the ALUs. The ALUs add/subtract the slices in parallel and the output is stored in one slice of the destination register. The next slices of the input operands

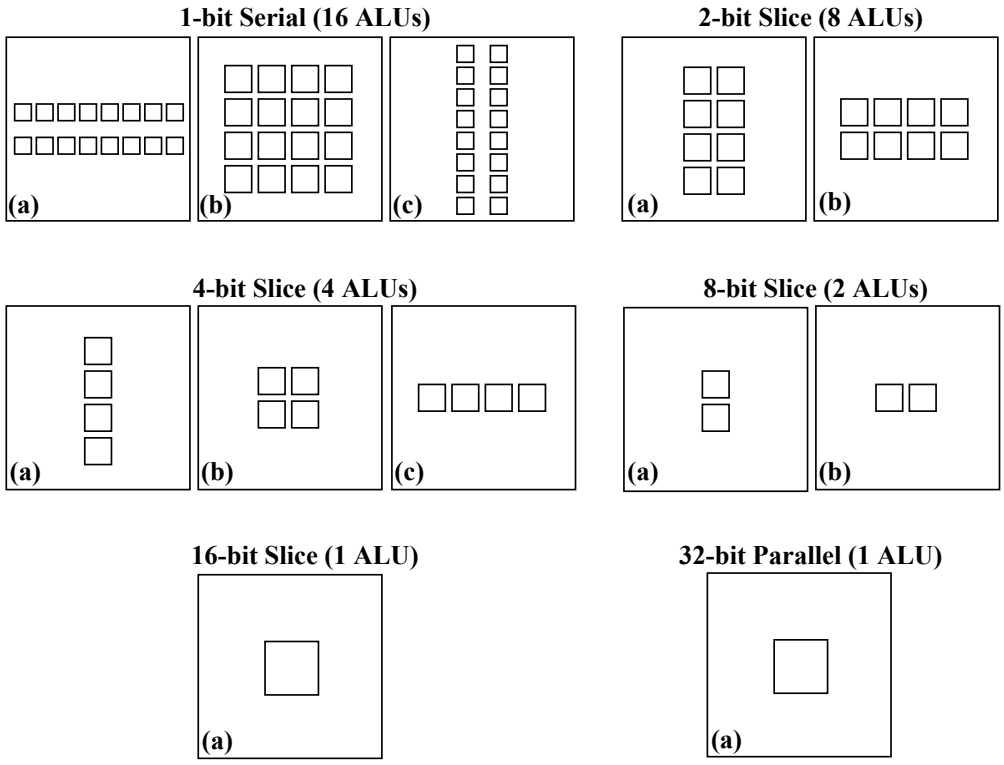

Figure 11: PE_blocks layout for bit-sliced methods.

are processed in a similar method until all the slices are processed. The carry-in or the borrow-in of each slice comes from the previous slice. In the bit-sliced methods, using an p-bit ALU for addition/subtraction of N-bit operands takes $3 \times \frac{N}{p}$ clock cycles. 3 clock cycles for reading one slice of the two operands, performing the arithmetic operation, and writing back the results for each slice. $\frac{N}{p}$ slices are sequenced through the p-bit ALUs to produce the final result. The Dual-Port BRAM allows both input operands to be read at the same time (one p-bit slice per clock cycle). The 32-bit parallel addition/subtraction requires one cycle to read the two full-precision operands from the register file, one clock cycle for performing the operation, and the third cycle to write the result back into the register file. The bit-sliced *Move* instructions follow the same method except that at each iteration, two slices from the source register is read from the register file and are written back into the BRAM of the destination register. The latency of *Move* instructions is also based on utilizing the Dual-Port BRAM, which is capable of reading and writing 2 slices at a time. This results in $\frac{N}{2p}$ read iterations and $\frac{N}{2p}$ write iterations on the total

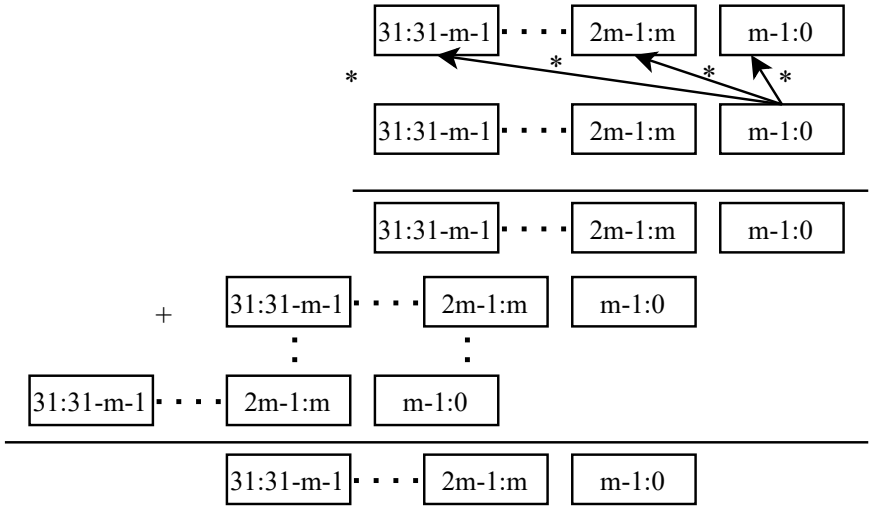

Figure 12: Bit-sliced multiplication algorithm.

N bits with accessing 2 slices at each clock cycle resulting in $\frac{N}{2p} \times 2 = \frac{N}{p}$ total clock cycles for *Move* instructions. The bit-sliced multiplication algorithm is shown in 12. Every two slices of the operands are multiplied by each other and they are accumulated in a temporary register in the ALUs. If the inputs are m slices, the result would have $2m$ slices. The middle m slices are selected as the final output. The bit-sliced multiplication is implemented using the standard multiply-accumulate method by multiplying two slices at a time, spending $2 \times (\frac{N}{p})^2$ clock cycles for generating the partial products and $\frac{N}{p}$ clock cycles for the partial production accumulation step.

5.2.3 Latency Comparison

To evaluate the effects of utilizing bit-sliced methods, we ran the same ML benchmarks presented in Section **??** using the bit-sliced optimization. Figure 13 compares the inference latency of the parallel and the serial implementations of these benchmarks. In the parallel implementation, the operands data-width is equal to the slice size, while in the bit-serial implementation, the data-width is processed serially one bit at a time. It is shown in Figure 13 that in the parallel implementation, the latency is almost a fixed value regardless of the data-width. However, in the bit-serial implementation, the latency depends on the data-width. This is because of the nature of

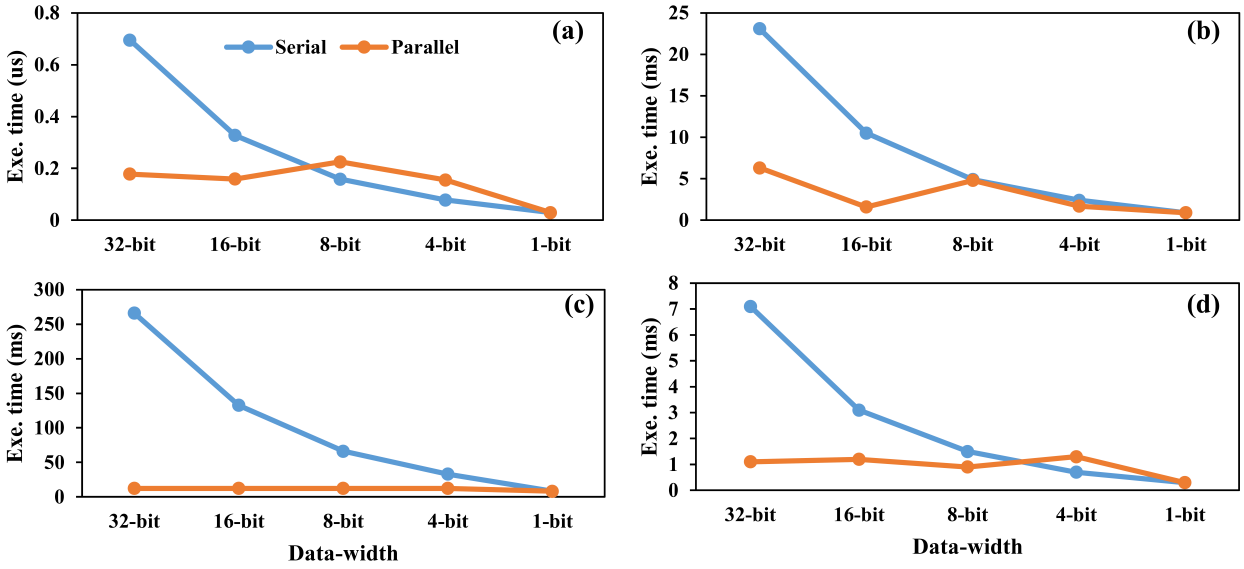

Figure 13: Benchmarks Execution time: (a) MLP, (b) LSTM, (c) CNN, (d) GRU.

bit-serial operations that the clock cycles spent for each instruction, depend on the data-width. This figure also shows that the gap between the serial and the parallel implementations decreases as the data-width decreases until 1-bit data-width that both methods result in the same latency.

The trade-off between the resource utilization and the instructions' clock cycle counts are evaluated using the *GOPS* metric that depends on both of these factors. Figure 14 shows the *GOPS* values for different ALUs' bit-widths at various data-widths. The results show that the bit-serial method outperforms all the other methods in addition/subtraction and the data movement operations. The bit-sliced methods vary in performance based on the instruction and the data-width. However, in bit-sliced methods, *GOPS* is often maximized when the data-width is equal to the slice size. For example, in 4-bit data width, the 4-bit sliced method results in higher *GOPS* than the rest of slice sizes. The diagrams also show that the *GOPS* values increase with the decreased data-width. It is because in lower data-widths, less number of clock cycles is spent per instruction. In the 32-bit Parallel and the 16-bit sliced implementations, the *GOPS* does not vary based on the data-width since the slice size is larger than all the data-widths. In these cases, the operation is performed on one slice anyway, although the data-width is smaller is smaller than that slice size.

Table 16: Small MLP Benchmark Execution Time (μs) for Different Methods

Method	**Data-Width**				#
	32-bit	**16-bit**	**8-bit**	**4-bit**	**PEs**
1-bit Serial	44.80	17.60	7.84	3.92	12×12
2-bit Sliced	23.80	4.40	4.56	2.44	20×10
4-bit Sliced	9.44	3.76	2.36	1.46	10×10
8-bit Sliced	4.40	2.36	1.46	1.46	20×10
16-bit Sliced	2.36	1.46	1.46	1.46	10×10
32-bit Parallel	1.46	1.46	1.46	1.46	10×10

An analysis on how the bit-sliced methods affect on the latency of a small MLP benchmark is shown in Table 16. This table shows that when the ML benchmark is small enough that it could fit into the processor array, how the latency is affected by different bit-serial and bit-sliced methods. Because in this case the whole ML benchmark would fit into the processor array, there is no need to apply the divide-and-conquer method and, therefore, the effects of resource utilization and the processor array size is not included in the results since, in this table, we only focus on the latency. The results show that the latency of all methods has a linear relation with data-width. The lower data-widths result in lower latency since the instructions take less clock cycles to go over the bits of data. Comparing different bit-serial and bit-parallel methods, at a fixed data-width, shows that the latency decreases as the ALU's bit-width increases. In this case, when the slice sizes are larger, less number of iterations are required on the total data-width. The instructions spend less number of clock cycles and therefore the latency decreases. However, this would not be followed for all ML benchmarks since for larger benchmarks that the divide-and-conquer method is applied, the number of PEs and the processor array size would also affect on the total latency.

5.2.4 Resource Utilization Comparison

Table 17 compares the resource utilization of the bit-serial, bit-sliced and the bit-parallel implementations for a Virtex-7 and a Virtex Ultra FPGA. This table also compares the maximum number of PEs that fit tnto these FPGAs for each method. Based on this table, as expected, the higher the ALUs bit-width is, the more resources are utilized for each ALU. In the resource

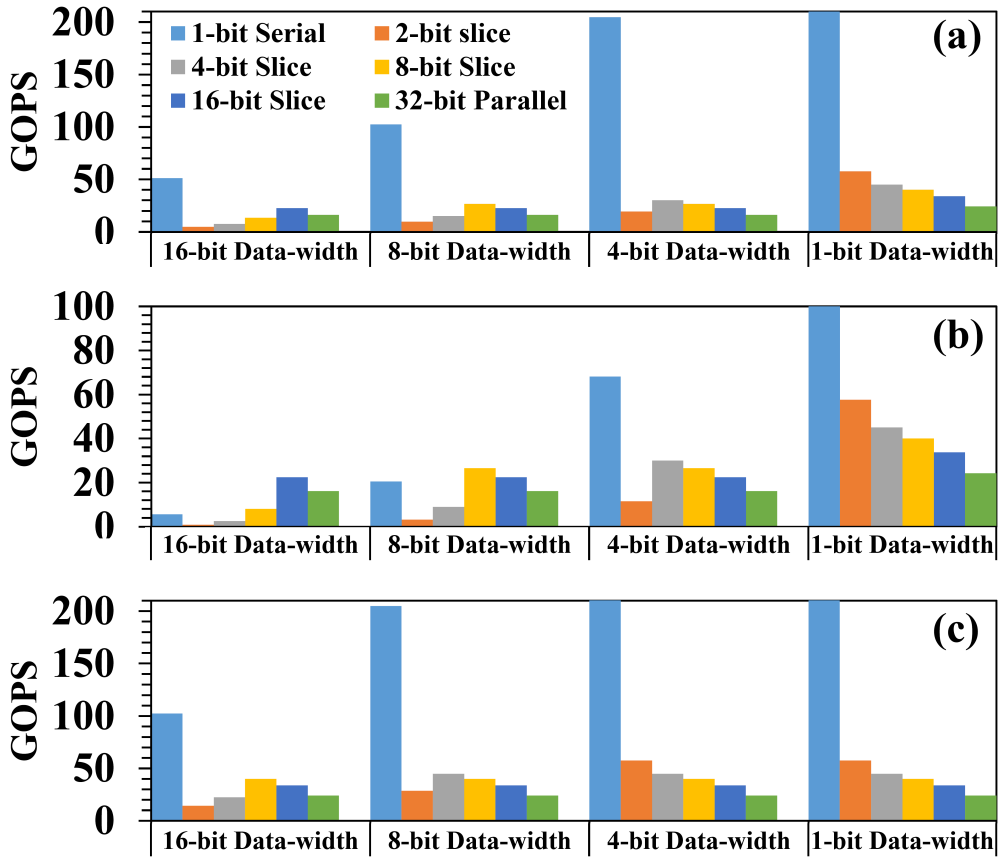

Figure 14: ALUs' Performance for Different Methods: (a) Addition/Subtraction, (b) Multiplication, (c) *NEWS* Moves.

utilization, it is shown that when the ALU's bit-width is 16 or 32 bits, the ALUs utilize DSPs in addition to the LUTs and FFs to implement the arithmetic operations. Using DSPs in the ALUs limits the total number of PEs to the number of available DSPs on each specific FPGA. The increased resource utilization from bit-serial to bit-sliced implementations comes from the more complex ALUs with more registers inside the ALUs. Comparing the LUT resource utilization in the 8-bit slice versus 16-bit slice shows that the LUT utilization has decreased from 127 to 73 whihc is because of the DSP utilization in the 16-bit slice methods. Therefore, the decreased LUT utilization has been compensated with the increased DSP utilization. The other comparison is between the two utilized FPGAs. The Virtex Ultra FPGA has around $4\times$ more LUTs than the Virtex-7 FPGA. This increased available resource results in higher number of PEs in the Virtex Ultra implementation. For example, for the bit-serial method, the total PEs change from $16k$ to $65k$ ($4\times$ more) when running the design on the Virtex Ultra FPGA.

The other metric that evaluates the resource utilization versus the instructions' latency is $Area \times Latency$ that is shown in Figure 15 for each of the processor array's instructions. Lower $Area \times Latency$ results in higher performance in the ALUs. It is shown that the bit-serial method provides the lowest values at all data-widths. The bit-sliced methods' performance increases as the data-width decreases. In case of 16-bit sliced and 32-bit parallel, the $Area \times Latency$ value is fixed for all data-widths since the data-width is less than the slice size, and the operations are conducted on one slice anyway, regardless of the data-width. Therefore, in these cases, the latency and the area are fixed. In all the bit-serial, bit-sliced, and bit-parallel methods, the area does not vary based on the data-width since for different data-widths, the ALU's architecture is fixed and it is the number of iterations that vary based on the data-width. However, the ALU's architecture changes based on the ALU's bit-width (slice size).

5.2.5 Performance Comparison

The functional density of the bit-serial, bit-sliced and bit-parallel implementations is shown in Figure 16. The functional density shows the level of concurrency in the SIMD architecture. It is

Table 17: Max number of PEs for Different Methods

ALU's bit-width	**ALU's Resources**			**Max**
	LUTs	FFs	DSPs	**Number of PEs**
Virtex-7				
1-bit Serial	2	2	0	$128 \times 128 = 16{,}384$
2-bit Slice	32	67	0	$48 \times 24 = 1{,}152$
4-bit Slice	73	69	0	$30 \times 30 = 900$
8-bit Slice	127	73	0	$40 \times 20 = 800$
16-bit Slice	73	81	1	$26 \times 26 = 676$
32-bit Parallel	105	97	4	$22 \times 22 = 484$
Virtex Ultra				
1-bit Serial	2	2	0	$256 \times 256 = 65{,}536$
2-bit Slice	32	67	0	$104 \times 52 = 5{,}408$
4-bit Slice	33	69	0	$66 \times 66 = 4{,}356$
8-bit Slice	42	73	0	$66 \times 33 = 2{,}178$
16-bit Slice	81	81	1	$45 \times 45 = 2{,}025$
32-bit Parallel	112	97	4	$40 \times 40 = 1{,}600$

computed by evaluating the number of parallel operations that can be performed on a fixed FPGA resources. For example, we limit the resource utilizaiton to $4k$ slices of the Virtex-7 FPGA and evaluate that how many parallel operations can be performed using that resources. The number of parallel operations are equal to the number of PEs within the processor array that fits into that limited resources. Higher functional density shows the method is more optimized in terms of reource utilization since more PEs could be packed into a fixed number of slices. As shown in Figure 16, the bit-serial method provides the highest density among all other methods. The less difference between the bit-serial and other methods at low area utilization, shows that how the bit-serial method is optimized for large benchmarks with low resource utilization compared to the bit-sliced or parallel methods. The gap between the bit-serial and other methods increase as more resources become available to the design. The values of the functional density for each method increase by increasing the number of available slices, but the rate of increase in the bit-serial method is more than the other methods. Among the bit-sliced methods, smaller slice sizes provide more concurrency by utilizing less resources such that 2-bit sliced has a higher functional density than 4-bit sliced and so on for the other slice sizes.

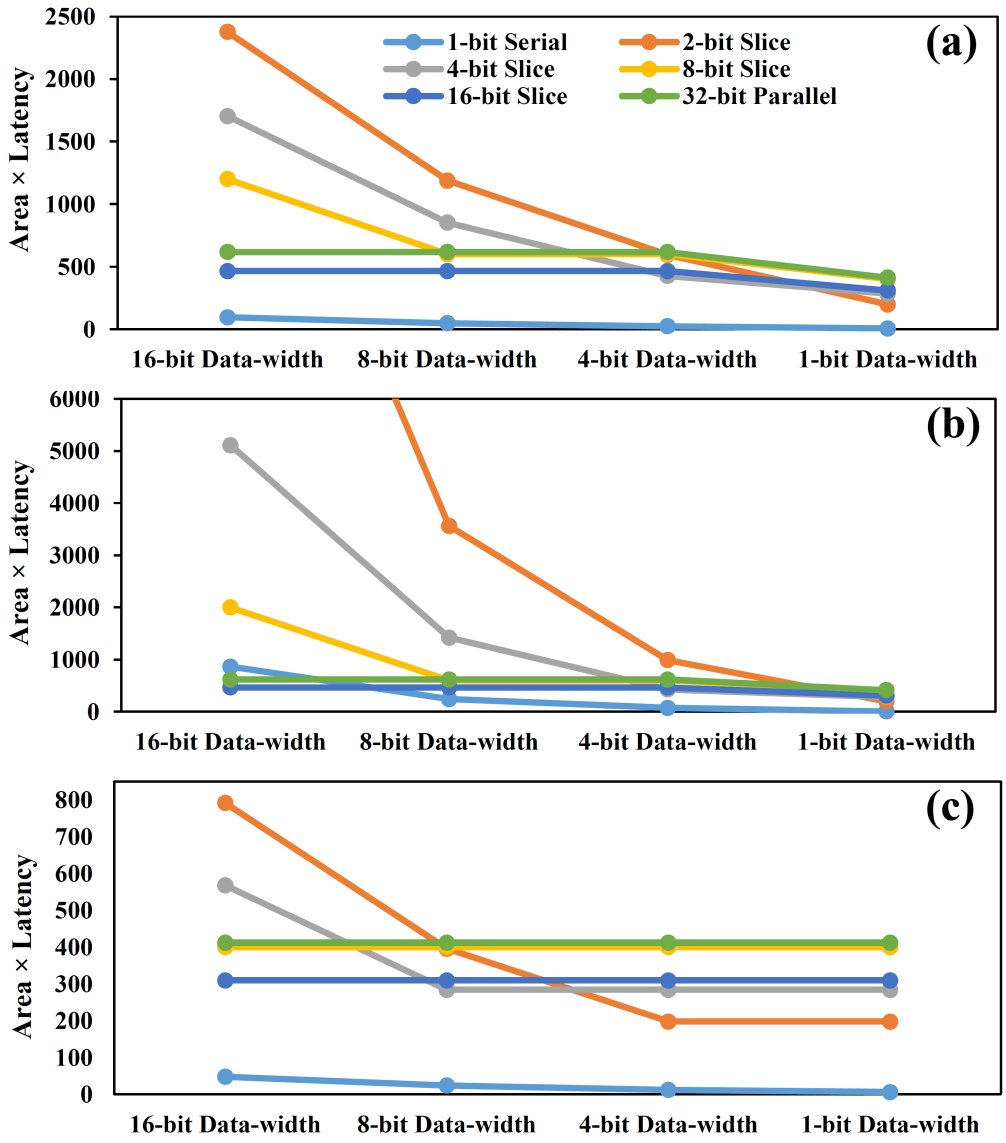

Figure 15: Area × Latency for Different Methods: (a) Addition/Subtraction, (b) Multiplication, (c) *NEWS* Moves.

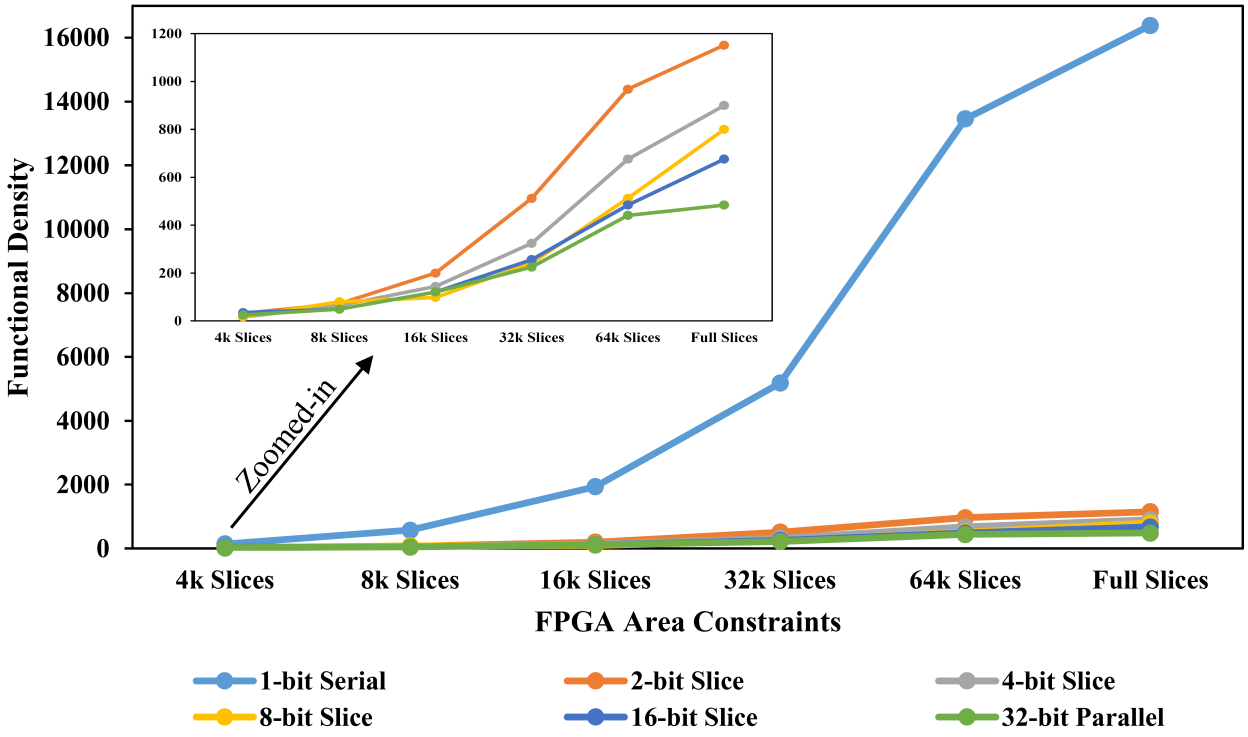

Figure 16: Functional Density for Different Methods.

To evaluate the effects of utilizing bit-sliced methods on the operating frequency of the processor array, we implemented the bit-serial and the bit-sliced methods with almost the same number of PEs (160 PEs) and compare the maximum operating frequency for each of these methods. The number of PEs in different designs could not be exactly the same since each has a different PE-block layout. For example, in the bit-serial method, the PE-blocks are 4×4 and the processor array could be squares with sizes that are multiples of 4. However, in the 2-bit sliced method, the PE-blocks size are 2×4 and therefore cannot support square size processor arrays. The maximum operating frequency of each processor array design is reported in Table 18. This table shows that the bit-serial method can work at a higher frequency than the bit-sliced and the bit-parallel methods. The small and simple bit-serial ALUs increase the performance of the design by a factor of 2 compared to the bit-parallel implementation. As the ALUs get larger in the bit-sliced methods, more resources are required to implement the same number of PEs, the FPGA gets congested and the design would have a lower performance in terms of clock frequency. However, in all the methods, when the processor array is larger, the operating frequency is

Table 18: Max Operating Frequency for Different Methods

Method	Freq. (MHz)	# PEs
1-bit Serial	180	20×20
2-bit Sliced	120	40×20
4-bit Sliced	120	20×20
8-bit Sliced	120	20×10
16-bit Sliced	110	20×20
32-bit Parallel	90	20×20

decreased. For example, for the bit-serial method, when the processor array size changes from 160 PEs to 16k PEs, the operating frequency changes from 180 MHz to 130 MHz. In our latency and area evaluation experiments, we compared the methods at the same clock frequency.

5.3 Online Training

Our SIMD processor array overlay was originally designed to accelerate the inference phase of ML applications. An extension of this design is then developed to support training of MLP networks on our FPGA overlay. Figure 17 shows the additional connections that are added to the original design to support back-propagation algorithm. The connection from the first row of PEs to the last column of PEs is implemented using a $Move\_N$ and then a $Move\_W$ instruction. The other connection from the last column of PEs to the first column of PEs is implemented using two $Move\_E$ instructions. In these connections, the data goes through the I/O buffers and then is stored to the destination PEs register files. The other modification is adding a *copy* instruction to the processor array's ISA that is used in the back-propagation path when computing the differential values from the hidden layer to the input layer. To add the *copy* instruction, the FSM-based design of the controller is modified. In the *copy* instruction, the data is copied from the source PEs' register to the destination PEs register on their right side. The number of copies is set as an input register. For example, (*copy* $R1$, $R2$, 4) instruction copies the $R2$ register of the first column to the $R1$ of the next 4 columns.

The back-propagation algorithms on a fully connected MLP networks is implemented by

computing the difference between the expected and the actual outputs and then updating the network parameters accordingly. After performing a forward path, the actual and expected outputs are subtracted based on (5) and then the output layer weights and biases are updated based on (6-(6)). Then the hidden layer network parameters are updated based on (8-11). In our overlay, each of these steps are implemented using SIMD instructions. The (5) is computed using an element-wise vector subtraction with the results stored in the last column of PEs. An element-wise vector multiplication that is followed by another element-wise vector subtraction is performed for bias updates. Two element-wise vector multiplication are performed to compute the (6) and another element-wise vector subtraction performs (7). Also, before performing the (6), *ColumnToColumn* software macro is called to copy the values computed in the last column (7) to the other columns to then be accessible for weight update (7). The same is applied when computing the (8-11) using element-wise vector subtraction and multiplication to updated the hidden layer weights and biases. In these equations, *out* is the neuron's output after applying the activation functions and *net* is the accumulated input values times the edge weights.

$$\frac{\partial E_{total}}{\partial out} = out - target \tag{5}$$

$$\frac{\partial E_{total}}{\partial W} = \frac{\partial E_{total}}{\partial out} \times \frac{\partial out}{\partial net} \times \frac{\partial net}{\partial W} \tag{6}$$

$$W^{'} = W - \alpha \times \frac{\partial E_{total}}{\partial W} \tag{7}$$

$$\frac{\partial E_{total}}{\partial W} = \frac{\partial E_{total}}{\partial out} \times \frac{\partial out}{\partial net} \times \frac{\partial net}{\partial W} \tag{8}$$

$$\frac{\partial E_{total}}{\partial out} = \Sigma \frac{\partial E_i}{\partial out} \tag{9}$$

$$\frac{\partial E_i}{\partial out_{h_i}} = \frac{\partial E_i}{\partial net} \times \frac{\partial net}{\partial out} \tag{10}$$

$$\frac{\partial E_i}{\partial net} = \frac{\partial E_i}{\partial out} \times \frac{\partial out}{\partial net} \tag{11}$$

To test the functionality of the MLP training algorithm, we ran some MLP benchmarks used in [126] dataset. Table 19 shows the results for the forward and backward iterations. In this table, the MLP size is the input size which represents the history size for the utilized dataset. The number of hidden nodes is 50 and the MLPs have one Boolean output which represents the state of the structure. The forward and backward latency is also shown in Table 19. The reported forward latency is the inference latency and the training latency would be the sum of one forward and one backward iteration (one epoch) times the number of epochs. The number of performed SIMD instructions for each MLP network is also presented. In the instruction count columns, the numbers on the left size of the plus sign are the number of instructions for the forward path and the numbers on the right side of the plus sign are the number of instructions for the backward path. The summation of these two is the number of instructions per epoch. Comparing these numbers show that the data movements between the PEs (*NEWS Moves* instructions) are the most used instructions. This is while the utilized overlay for these benchmarks is optimized for internal data movement using binary tree interconnection network. The resource utilization is also presented in this table. All benchmarks are ran on a processor array of $10k$ PEs and therefore, they share the same resource utilization. Since all of the benchmarks sizes are larger than the processor array, we still follow the divide-and-conquer method for implementing the back-propagation algorithms.

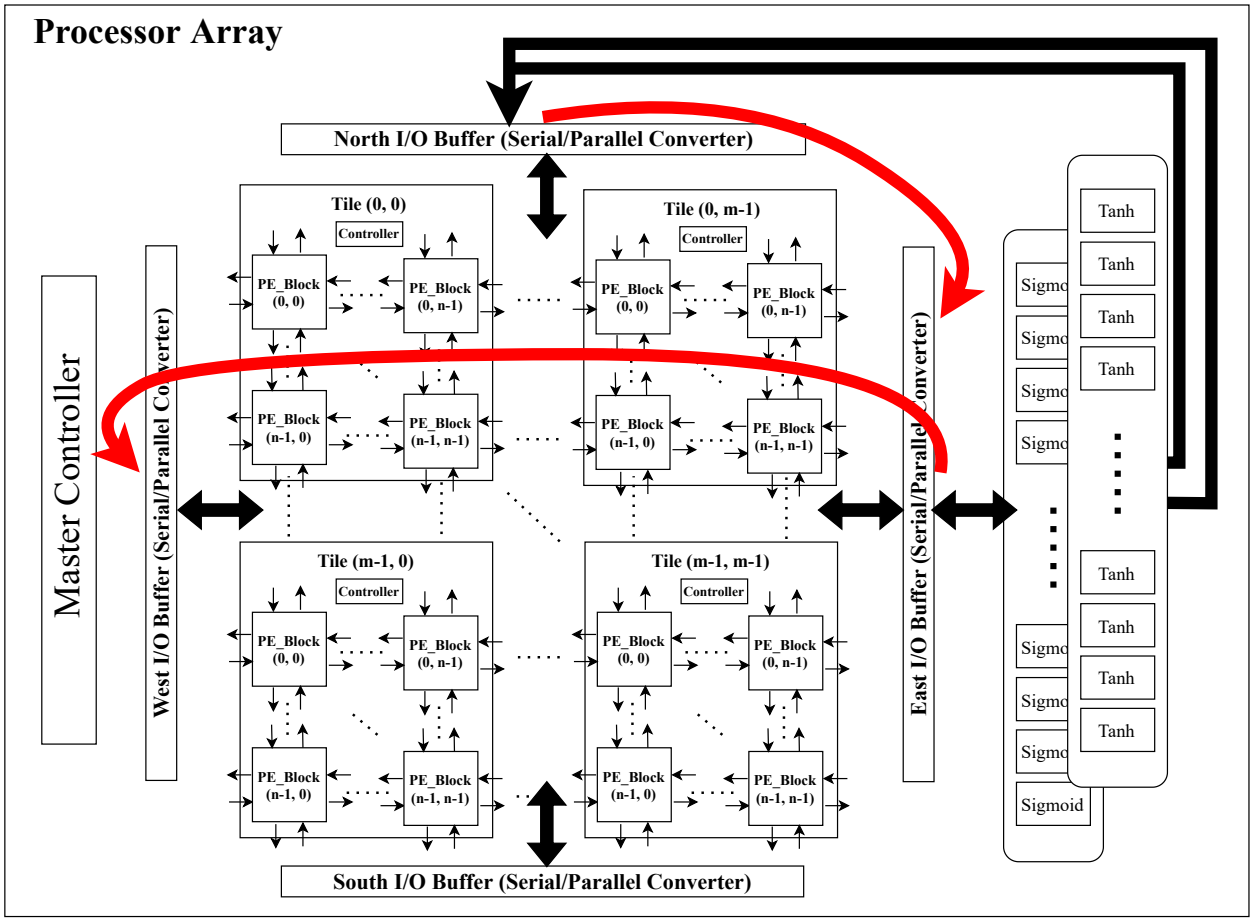

Figure 17: Modifications for Supporting Online Training.

Table 19: Processor Array Overlay Results (8-bit FxP, 200 MHz, Virtex UltraScale+ VU9P)

MLP	**Forward**	**Backward***	**Instruction Count****				**Resource Utilization*****				**#**
size	**Latency (μs)**	**Latency (μs)**	**ADD**	**SUB**	**MULT**	**MOVE**	**LUTs**	**FFs**	**BRAMs**	**DSPs**	**PEs**
200	8.76	7.60	36+7	0+6	3+9	117+74					
700	22.36	12.36	91+12	0+11	8+19	297+74					
1100	32.16	16.60	138+16	0+15	12+27	408+83	613220	83659	625	0	10k
1300	39.28	18.60	184+18	0+17	14+31	474+85	(51.8%)	(3.5%)	(14.4%)	(0%)	
2200	60.76	27.60	336+27	0+26	23+49	617+94					
3200	123.16	37.60	284+37	0+36	33+69	2181+104					

* For training, each epoch's latency would be the sum of one forward and one backward iteration.

** The numbers on the left side of the plus sign are the forward path instruction count and the right side numbers are the backward path instruction count.

*** All MLP networks are run on the same processor array and therefore have the same resource utilization.

5.4 Design Space Exploration

Our overlay can be configured in various data-widths, various processor array sizes, various ALU's bit-width, on various FPGAs. Thus, the design space contains a number of parameters that can be adjusted based on the application. The design space is large due to numerous design choices and therefore it takes a long time to synthesize all these possible architectures and find the best processor array overlay architecture for a specific ML application. Moreover, it is hard to provide a single efficient hardware solution for an end-to-end FPGA implementation of every different ML application. In this section, we present an analytical analysis on how utilizing different parameters affected on the inference latency of ML applications. We have explored these different configurations using a set of equations that predict the inference latency for different overlay architectures and can be used to find the optimum design for a specific ML application. Although because of the programability and the flexibility that our design provides, different ML applications could still be ran on the overlay, but they are not just as optimized. These equations can be solved to determine design parameters that can be used to set data-width and ALU bit-widths for any LSTM/GRU/MLP network and customize the overlay for the target neural network model. The flow shown in Figure 18 combines the results of the device specific ALU sizing analysis with network specific information within the set of parametric equations shown in (12-17). (12-14) are for LSTM networks and (15-17) are for MLP networks. We have also developed equations for GRU networks but as they are sufficiently similar to the LSTM equations (different constant factors) that we have omitted them. These equations can be used in a straight forward manor to compute the total clock cycles spent for each of the three major instruction types: multiplication, addition, and data movement on a specific configuration of the 2-D SIMD processor array. This allows application developers to predict the end-to-end inference latency of their particular ML application on a specific device early on in the design cycle prior to synthesis. The need to build and perform trial and error explorations of building systems and simulating their application running on varying combinations of architecture design choices for a specific device. The total predicated inference latency in clock cycles on our 2-D SIMD processor array is

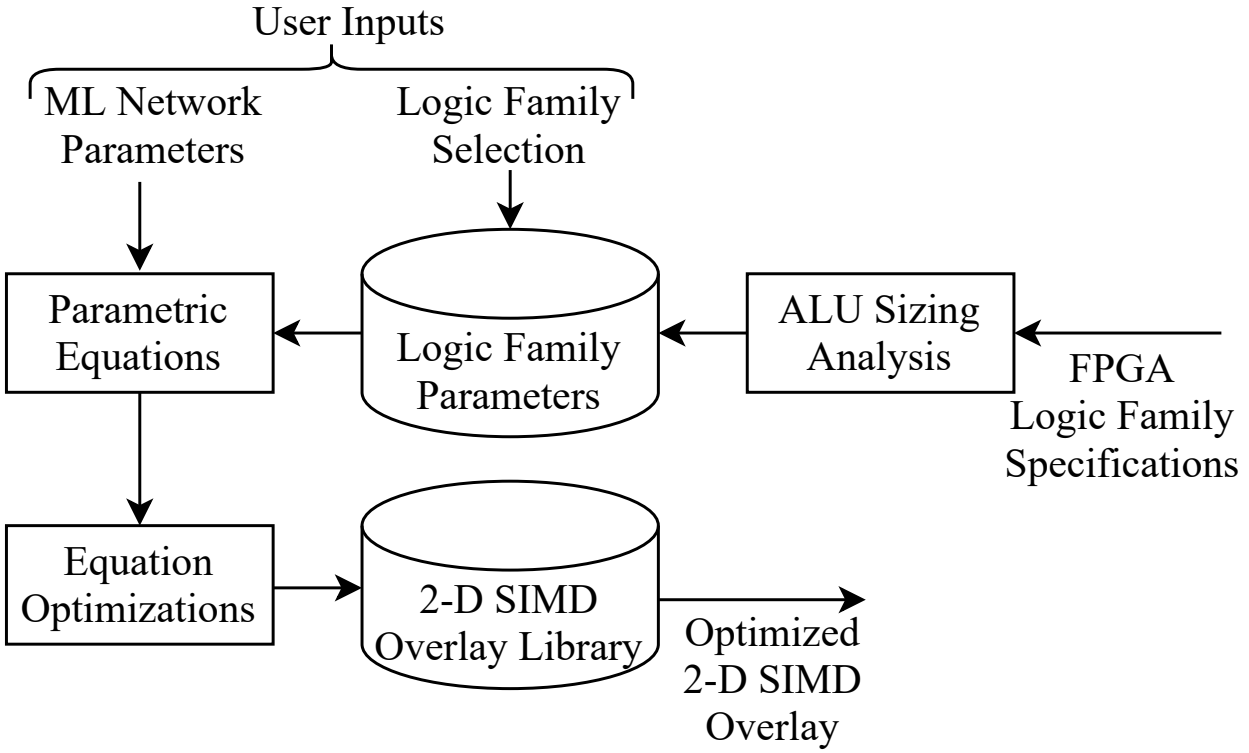

Figure 18: SoC with Overlay.

the sum of the three equations. Conversely, the equations can be solved to determine the optimum values of key design parameters by differentiating with respect to that parameter.

$$LSTM\_Adds\_Total\_Cycles = (\sum_{i=1}^{n-2} \frac{4m_{i+1}(m_i(m_i - 2^{\log_2 m_i} + \log_2 m_i + 2)) + (m_{i+1}(m_{i+1} - 2^{\log_2 m_{i+1}} + \log_2 m_{i+1} + 2))}{x_i^{'} y_i^{'}} + (\sum_{i=1}^{n-1} \frac{9m_i}{y_i^{'}}) + \frac{m_n m_{n-1}(m_{n-1} - 2^{\log_2 m_{n-1}} + \log_2 m_{n-1} + 2)}{x_i^{'} y_i^{'}} + \frac{m_n}{y_i^{'}}) \times ((2+q)\frac{N}{p}) \tag{12}$$

$$LSTM\_Mults\_Total\_Cycles = (\sum_{i=1}^{n-2} \frac{4(m_{i+1}m_i + m_{i+1}^2)}{x_i^{'} y_i^{'}} + \sum_{i=1}^{n-1} \frac{3m_i}{y_i^{'}} + \frac{m_n m_{n-1}}{x_i^{'} y_i^{'}}) \times ((q+1)\frac{N}{p}^2 + (2-q)\frac{N}{p}) \tag{13}$$

$$LSTM\_Moves\_Total\_Cycles = (\sum_{i=1}^{n-2} \frac{4x(m_{i+1}m_i + m_{i+1}^2)}{x_i^{'} y_i^{'}} + \sum_{i=1}^{n-1} \frac{14m_i}{y_i^{'}} + \sum_{i=1}^{n-1} 2m_{i+1} + \frac{x m_n m_{n-1}}{x_i^{'} y_i^{'}}) \times (\frac{N}{p}) \tag{14}$$

$$MLP\_Adds\_Total\_Cycles = (\sum_{i=1}^{n-2} \frac{m_{i+1}(m_i(x_i^{'} - 2^{\log_2 x_i^{'}} + \log_2 x_i^{'} + 2))}{x_i^{'} y_i^{'}} + \frac{m_i}{y_i^{'}}) \times ((2+q)\frac{N}{p}) \tag{15}$$

$$MLP\_Mults\_Total\_Cycles = (\sum_{i=1}^{n-2} \frac{(m_{i+1}m_i)}{x_i^{'} y_i^{'}}) \times ((q+1)\frac{N}{p}^2 + (2-q)\frac{N}{p}) \tag{16}$$

$$MLP\_Moves\_Total\_Cycles = (\sum_{i=1}^{n-2} \frac{(x-1)(m_{i+1}m_i)}{x_i^{'} y_i^{'}} + \sum_{i=3}^{n-1} m_{i+1}) \times (\frac{N}{p}) \tag{17}$$

5.4.1 Defining the Equations

Table 20 provides definitions for the parameters used in our equations. The first four parameters are user defined. These include the ML network parameters (m_i for the number of nodes in the i_{th} layer of the network and n for total layers in the network), the operand precision (listed as data-width N), and ALU arithmetic precision (p as the slice bit-width). x, y are the 2-D processor array dimensions (number of PEs) input from the prior ALU sizing analysis. q is a boolean variable that selects between bit-serial and non bit-serial methods. 0 selects bit-serial and 1 for bit-sliced and bit-parallel methods. $x_i^{'}, y_i^{'}$ are division factors for large matrix-vector operations. These are set for ML networks that cannot fit into the processor array. In such cases, a divide-and-conquer method is employed that divides the operations into sub matrices of size $x_i^{'}, y_i^{'}$ that can fit into the processor array. When the dimensions of the neural network (m_i, m_{i+1}) fit into the processor array (i.e, m_i, m_{i+1} are smaller than x, y), $x_i^{'}, y_i^{'}$ are equal to m_i, m_{i+1}. Conversely, when the dimension of the neural network (m_i, m_{i+1}) exceeds x, y, it is divided into smaller subsections. In this case, $x_i^{'}, y_i^{'}$ are equal to the largest factors of m_i, m_{i+1} that are smaller than x and y. For example, if the 2-D processor array size is $x = 60$ $y = 60$ and the network dimension is $m_i = 100$, $m_{i+1} = 90$, then the largest factors of m_i and m_{i+1} that are smaller than x, y are 50 and 45. Therefore, $x_i^{'} = 50$ and $y_i^{'} = 45$.

5.4.2 Explaining an Example Equation

As a representative example, we provide a description of (12) which computes the number of cycles for the *add* operations in the LSTM networks. For refreshing on the LSTM architecture and technical terms refer to [127]. The provided equation includes two terms. The first term computes the instruction counts which is then multiplied by the second term which is the clock cycles per addition instruction. The first term in the equation is composed of the two summations and two single terms which is then multiplied by the second term $(2+q)\frac{N}{p}$. Expanding on the first term, the first summation is for the number of additions of the LSTM layers and operate over each of the $n-1$ layers of the network. In the numerator of this summation, the first term is for the

additions within the matrix-vector multiplication of normal weights and the second term is for the recurrent weights. The next summation is resulted from the vector-vector addition operations within each LSTM gate, when adding the outputs of normal weights, recurrent weights and the biases. The next single term (without the summation) is for the additions of matrix-vector operation for the fully connected layer and the last single term is for adding the biases for that fully connected layer. The constant factor of 4 in the numerator of the first summation is because an LSTM network has 4 gates that perform the same operations. The terms with log_2 in the numerators resulted from the utilized binary tree reduction method for partial product accumulation. In case the hidden layer size is a power-of-two, all accumulations are performed using binary tree, and in case of a non power-of-two hidden layer size, the binary tree reduction is performed on the largest power-of-two and the rest of accumulations is implemented using linear addition. The *ceiling* and $flooring$ operators are taking care of these cases. All the denominators of these terms determine the division factors. For instance, back to our example in Section 5.4.1, when $m_n = 90$ and $y_i^{'} = 45$ the vectors are divided into $\frac{90}{45} = 2$ subsections and therefore we perform 2 SIMD element-wise additions for the network bias addition in the fully connected layer that is shown by $\frac{m_n}{y_i^{'}}$ in (12). The values of the second term for each equation are derived based on the cycle counts per instruction shown in Table 15. For instance, in (12), the second term $((2+q)\frac{N}{p})$ is the cycles per addition. This is derived based on the values of the "Addition/Subtraction" column of Table 15. Looking at this term, replacing the $q = 0$, $p = 1$ results in the addition/subtraction cycles for the bit-serial method (shown in Table 15 with $2N$), replacing the $q = 1$, $p = p$ results in $3\frac{N}{p}$ for the bit-sliced methods, and replacing the $q = 1$, $p = 32$, $N = 32$ results in 3, equal to what is shown in Table 15 for the 32-bit parallel method. Similarly, Table 15 lists the values used as the second term for *mult* and *Move* instructions.

5.4.3 Mapping to SIMD Instructions

Table 21 shows a summary of instruction counts per software macro. This table shows the instructions for matrix-vector operations when the ML network size is small and fits into the

Table 20: Parameter Definition of Cycle Count Equations.

Parameter	Description	Type
N	Data-width	User-defined
m_i	Number of nodes for the i_{th} layer	User-defined
n	Number of layers	User-defined
p	ALUs' bit-width (slice size)	User-defined
q	Binary variable to switch between bit-serial and other methods	Automatically set based on p value
$x_i^{'}, y_i^{'}$	Division factor for the i_{th} layer	Board-specific
x, y	Max. number of PEs	Board-specific

processor array. In this case, an MVM operation performs 1 multiplication to generate the partial products, $\log_2 m_i$ addition operations to accumulate the partial products using a binary tree reduction method, and $x\ Move\_E$ operations to move the partial products between the PEs to be accumulated. With our SIMD architecture, only one multiplication per matrix-vector multiply is executed and the partial-product accumulation is performed in parallel on the matrix rows (m_{i+1}). For vector-vector operations (VVA and VVM), a single add/multiply instruction is performed on the vector elements in parallel. Therefore, for a VVA or a VVM operation, only 1 addition or 1 multiplication is performed. For matrix-vector operations that the network size is too large to fit into the processor array, the MVM operation is divided into $(\frac{m_i}{x_i^{'}} \times \frac{m_{i+1}}{y_{i+1}^{'}})$ subsections. For these cases, we have $(\frac{m_i}{x_i^{'}} \times \frac{m_{i+1}}{y_{i+1}^{'}})$ subsections where each subsection is a small MVM operation. Therefore, the number of instructions for each MVM operation would be multiplied by $(\frac{m_i}{x_i^{'}} \times \frac{m_{i+1}}{y_{i+1}^{'}})$. For VVA and VVM operations, the vector is divided into $\frac{m_{i+1}}{y_{i+1}^{'}}$ subsections. To compute the total instructions for an ML algorithm, the instruction counts per software macro are multiplied by the number of software macros utilized to implement an ML algorithm. For instance, each gate in an LSTM network, that is computed by $(WX + UH + b)$ [127], includes two MVM operations and two VVA operations. All operations are accounted for in the equations shown in (12-17).

Table 21: Software Macros Instruction Count Break-Down.

Instruction	**Software Macro**		
	MVM $(m_{i+1} \times m_i)$	VVA (m_i)	VVM (m_i^*)
add	$\log_2 m_i$	1	0
$mult$	1	0	1
$NEWS$ $Moves$	x	0	0

* Numbers in parenthesis are the matrix/vector sizes.

$$LSTM\_Adds\_Total\_Cycles = (\sum_{i=1}^{1} \frac{4\times 6(4(4-2^{\log_2 4}+\log_2 4+2))+(6(6-2^{\log_2 4}+\log_2 6+2))}{4\times 2} + (\frac{9\times 4+9\times 6}{2}) + \frac{2\times 6(6-2^{\log_2 6}+\log_2 6+2)}{3\times 2} + \frac{2}{2}) \times ((2+1)\frac{32}{2}) = (234)\times(48) \tag{18}$$

$$LSTM\_Mults\_Total\_Cycles = (\sum_{i=1}^{1} \frac{4(4\times 6+6^2)}{2\times 4} + \frac{3\times 4}{2} + \frac{3\times 6}{2} + \frac{2\times 6}{2\times 3}) \times ((1+1)\frac{32}{2}^2 + (2-1)\frac{32}{2}) = (47)\times(528) \tag{19}$$

$$LSTM\_Moves\_Total\_Cycles = (\sum_{i=1}^{1} \frac{4\times 4(4\times 6+6^2)}{4\times 2} + \frac{14\times 4}{2} + \frac{14\times 6}{2} + (2\times 6+2\times 2) + \frac{4\times 6\times 2}{3\times 2}) \times (\frac{32}{2}) = (214)\times(16) \tag{20}$$

5.4.4 Verifying the Equations

For example purposes, we validated the LSTM equations on a small network and compare the instruction counts resulted from our equations versus the accurate cycle counters in the implemented design. Our example LSTM includes an input layer, a hidden layer and a fully connected layer at the output with sizes of 4, 6, and 2, respectively. Assuming the bit-sliced size is $p = 2$ and the processor array size of $x = 4, y = 2$. Using this information, we first compute the $x_i^{'}, y_i^{'}$ for each layer. From the input to the hidden layer we have $m_i = 4, m_{i+1} = 6$, and from hidden layer to output layer, $m_{n-1} = 6$, $m_n = 2$. Considering the processor array size ($x = 4, y = 2$), $x_1^{'} = 4$ because 4 is the largest factor of $m_1 = 4$ that is smaller than $x = 4$. $y_1^{'} = 2$ since 2 is the largest factor of $m_2 = 6$ that is smaller than $y = 2$. Similarly, $x_2^{'} = 3$ because 3 is the largest factor of $m_2 = 6$ that is smaller than $x = 4$. Finally, $y_2^{'} = 2$ because 2 is the largest factor of $m_2 = 6$ that is smaller than $y = 2$. These values are replaced in (12-14) to count the instructions using our

equations. They result the same instruction counts we get from implementing this sample network on our processor array. Based on these numbers, the total cycles for the inference phase of this LSTM network would be 234×48 cycles for additions, 47×528 cycles for multiplications, and 214×16 cycles for the *NEWS Moves* when the slice size is 2 and data-width is $N = 32$. We can replace the p parameter with other slice slices for computing the total cycles spent using the other ALU's bit-widths.

Chapter 6

Conclusion

This dissertation developed a memory-centric FPGA overlay which is designed using a SIMD processor array to support rapid coding, programmability, and flexibility for acceleration of ML applications. Our FPGA-based overlay has the potential to bring software levels of productivity and code portability to ML applications running on different FPGA devices. The developed architecture is based on a 2-D array of processor-in-memory tiles each consisting of small bit-serial ALUs with concurrent access to register files. In the our memory-centric architecture, every couple of PEs share the same BRAM block as their register file and therefore can access the other PEs within the same PE-block. This architecture eliminates the physical connections between the PEs and increases the level of concurrency in the design that is provided with our SIMD architecture. Experimental results showed that our SIMD array of the bit-serial processor-in-memory tiles outperformed custom full-precision HLS-based designs in some standard ML benchmarks. Importantly, a single hardware design was used for all performance comparisons, with the different networks being implemented through compiling software without requiring re-synthesis. The same code was run without modification on the system implemented in a different device. The bit-serial nature of the design and the novel method of storing the values within the PEs register files provided the opportunity to run various data-widths on the same hardware design.

Different optimizations are implemented on the original overlay to further reduce the latency. Our ML benchmark evaluation showed that in communication-bound networks such as MLP/LSTM/GRU the most time consuming part of the inference algorithm is spent on internal data movement between the PEs. The interconnection network is optimized from the standard *NEWS* method to binary tree method. The results showed a 33.3%, 28.0%, and 42.8% improvement on the latency of the utilized MLP, LSTM and GRU networks. However, this optimization comes at the cost of 268.1% and 36.1% increased resource utilization in LUTs and

FFs. Additionally, we also explored using bit-sliced and bit-parallel methods instead of the bit-serial operations. In this case, the ALU's bit-widths are more than one bit and therefore the ALU's resource utilization increases in terms of LUTs and FFs. The decreased latency of instructions in the bit-sliced methods is resulted from processing more bits in parallel (each slice) and spending less iterations on the operands data-widths. In this method, we processed a number of bits (one slice) in parallel and connect the slices serially to perform the operation on the whole data-width. The ALUs resource utilization was increased, but the instructions' latency and the total inference latency was decreased. The other optimization is supporting the training phase of the MLP applications. We modified the original design by adding more connections between the PEs and adding another SIMD instruction to the processor array's ISA. With the optimized design, we performed back-propagation algorithms for the MLP networks. This is tested on a real-time application and the results showed that the processor array meets the real-time requirements for most of the benchmarks. The 24.4% and 9.3% increase in the LUTs and FFs utilization is resulted by supporting the training phase on the processor array. As the processor array's architecture depends on a number of parameters including various data-widths, ALU bit-widths, 2-D processor array's size, and the utilized FPGA. We also explored the design space using a set of equations that predicted the ML inference latency before synthesizing. Using our set of equations, we find the optimum design architecture for the processor array for every ML application. However, the processor array would still run different ML applications but they not just as optimized.

Future work will explore developing front ends for standard machine learning domain-specific languages such as tensor-flow, and more exhaustive analysis and optimization on the latency and performance of the design. Developing a framework to fully automate the process of compiling ML applications into the processor array architecture is also an area of interest. A customization that can be applied to the processor array's PEs is to utilize a subsection of PEs for some specific algorithms and software macros. In this case, some PEs would be idle and can be switched off to be inactivated and save on dynamic power consumption. This would also be

helpful in terms of latency in case the application size is small that can operate on a subsection of PEs. The clock cycles spent on internal data movement would be saved since the data does not need to go through the inactive PEs. The other aspect of further optimizations can focus on large ML applications. In this case, the available processor arrays are not enough to fit the whole matrix/vectors at once. We currently follow a divide-and-conquer method for these benchmarks which increases the latency. The other method that can be implemented using dedicating the BRAM's register files to more PEs that are called virtual PEs. For example, in this method, instead of connecting 16 PEs to a BRAM block of size 16×1024 bits, we can connect 32 PEs to the same size BRAM and decrease each PE's local storage from 1024 bits to 512 bits. Therefore, by supporting more PEs, the level of concurrency increases and larger benchmarks can fit into the available physical and virtual PEs. Another area of interest would be to analyse the design's performance in terms of operating frequency. The current investigations on the limiting factors of the design's performance is the BRAM and ALUs connections. The critical path is all ALUs of a PE-block access the same BRAM block for data reads and writes. A solution that we have tried is to put registers withing the BRAM-ALU path to be able to clock the ALUs faster. However, this method increases the instruction's clock cycles since a clock cycle is added to every bit read. The improvement in the operating frequency is canceled out with the instructions clock cycle count that resulted in total higher latencies. This approach could be improved by using a pipeline architecture when accessing the BRAM. In that case, the data is read from the BRAM still one at a clock cycle, however it would be processed by the ALUs within the next clock cycles when the data is passed from the BRAM to the ALUs after going through the middle registers. The optimization can be in the connection between the PS and PL side of the whole design and find an alternative solution for a shared memory region between the custom IP and the MicroBlaze processor. The current solution uses AXI slave registers for this connection method, which is not very efficient and has limited the operating frequency. Implementing other ML application's training algorithms is also an area of interest as a future work. The other future work could be optimizing the scheduler (software macros) to support more concurrency and efficient processor

utilization for all ML benchmarks, specifically the CNN networks.

References

[1] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, et al., "Why compete when you can work together: Fpga-asic integration for persistent rnns," *In 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 199–207, 2019.

[2] A. Putnam, A. M.Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, et al., "A reconfigurable fabric for accelerating large-scale datacenter services," *In 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 13–24, 2014.

[3] P. Bannon, G. Venkataramanan, D. D. Sarma, and E. Talpes., "Computer and Redundancy Solution for the Full Self-Driving Computer'," *IEEE Hot Chips 31 Symposium (HCS)*, 2019.

[4] https://training.ti.com/jacinto7

[5] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, et al., "Think fast: a tensor streaming processor (TSP) for accelerating deep learning workloads," *In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 145–158, 2020.

[6] A. Ishfaq. "Discover Internet of Things editorial," *Discover Internet of Things*, 10.1007/s43926-021-00007-6, 2021.

[7] https://www.microsoft.com/en-us/research/project/project-catapult/

[8] A. Panahi, E. Kabir, A. Downey, D. Andrews, M. Huang, J. D. Bakos, "High-Rate Machine Learning for Forecasting Time-Series Signals," *IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2022. (Accepted, Not Published yet)

[9] H. Liu, A. Panahi, D. Andrews, and A. Nelson, "An FPGA-Based Upper-Limb Rehabilitation Device for Gesture Recognition and Motion Evaluation Using Multi-Task Recurrent Neural Networks," *IEEE Sensors Journal*, 2022.

[10] A. Panahi, S. Balsalama, A. T. Ishimwe, J. M. Mbongue, and D. Andrews, "A Customizable Domain-Specific Memory-Centric FPGA Overlay for Machine Learning Applications," *In 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 24–27, 2021.

[11] H. Liu, A. Panahi, D. Andrews, and A. Nelson, "FPGA-based gesture recognition with capacitive sensor array using recurrent neural networks," *In IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 225–225, 2020.

[12] S. Basalama, A. Panahi, A. T. Ishimwe, and D. Andrews, "SPAR-2: A SIMD Processor Array for Machine Learning in IoT Devices," *In 3rd International Conference on Data Intelligence and Security (ICDIS)*, pp. 141-147, 2020.

[13] D. Capalija, and T. Abdelrahman, "A coarse-grain fpga overlay for executing data flow graphs," *In The Second Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL)*, 2012.

[14] Z. Aklah, and D. Andrews, "A Flexible Multilayer Perceptron Co-processor for FPGAs," *In International Symposium on Applied Reconfigurable Computing*, pp. 427–434, 2015.

[15] A. M. Abdelsalam, F. Boulet, G. Demers, J. P. Langlois, and F. Cheriet, "An efficient FPGA-based overlay inference architecture for fully connected DNNs," *In 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–6, 2018.

[16] D. H. Noronha, R. Zhao, Z. Que, J. Goeders, W. Luk, and S. Wilton, "An Overlay for Rapid FPGA Debug of Machine Learning Applications," *In 2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 135–143, 2019.

[17] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2018.

[18] M. S. Abdelfattah, D. Han, A. Bitar, R. DiCecco, S. O'Connell, N. Shanker, et al., "Dla: Compiler and fpga overlay for neural network inference acceleration," *In 2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 411–4117, 2018.

[19] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W. M. Hwu, and D. Chen, "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs," *In 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2018.

[20] Y. Yu, T. Zhao, K. Wang, and L. He, "Light-OPU: An FPGA-based Overlay Processor for Lightweight Convolutional Neural Networks," *In The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 122–132, 2020.

[21] R. Shi, Y. Ding, X. Wei, H. Li, H. Liu, H. K. H. So, and C. Ding, "FTDL: a tailored FPGA-overlay for deep learning with high scalability," *In 2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2020.

[22] L. Ioannou and S. A. Fahmy, "Neural Network Overlay Using FPGA DSP Blocks," *In 2019 29th International Conference on Field Programmable Logic and Applications (FPL)* pp. 252–253, 2019.

[23] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "OPU: An FPGA-Based Overlay Processor for Convolutional Neural Networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[24] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, et al., "In-datacenter performance analysis of a tensor processing unit," *In Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

[25] X. Wang, V. Goyal, J. Yu, V. Bertacco, A. Boutros, E. Nurvitadhi, et al., "Compute-Capable Block RAMs for Efficient Deep Learning Acceleration on FPGAs," *In 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 88–96, 2021.

[26] O. Cetin, F. Temurtas, and S. Gulgonul, "An application of multilayer neural network on hepatitis disease diagnosis using approximations of sigmoid activation function," *Dicle Medical Journal/Dicle Tip Dergisi*, vol. 42, no. 2, 2015.

[27] Chad Greene (2020). sigmoid (https://www.mathworks.com/matlabcentral/fileexchange/51007-sigmoid), MATLAB Central File Exchange. Retrieved January 29, 2020.

[28] H. Amin, K.M. Curtis, and B.R. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," *IEEE Proceedings-Circuits, Devices and Systems*, vol. 144, no. 6, pp. 313–317, 1997.

[29] S. Gomar, M. Mirhassani, and M. Ahmadi, "Precise digital implementations of hyperbolic tanh and sigmoid function," *In 50th Asilomar Conference on Signals, Systems and Computers*, pp. 1586–1589, 2016.

[30] S. Khan, H. Rahmani, S. A. A. Shah, and M. Bennamoun, "A guide to convolutional neural networks for computer vision," *Synthesis Lectures on Computer Vision*, vol. 8, no. 1, pp. 1–207, 2018.

[31] A. Graves, A. R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," *In 2013 IEEE international conference on acoustics, speech and signal processing*, pp. 6645–6649, 2013.

[32] J. S.. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett, and N. L. Dahlgren, "DARPA TIMIT Acoustic Phonetic Continuous Speech Corpus CDROM NIST," 1993.

[33] Y. Guan, Z. Yuan, G. Sun, and Cong, J., "FPGA-based accelerator for long short-term memory recurrent neural networks," *In 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 629–634), 2017.

[34] https://github.com/karpathy/char-rnn

[35] https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt

[36] A. X. M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on FPGA," *arXiv preprint arXiv:1511.05552*, 2015.

[37] W. uo, H. E. Yantir, M. E. Fouda, A. M. Eltawil, and K. N. Salama, "Toward the Optimal Design and FPGA Implementation of Spiking Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[38] Y. LeCun, and C. Cortes, “MNIST handwritten digit database,” 2010.

[39] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J.Dally, and K. Keutzer, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size,“ *arXiv preprint arXiv:1602.07360*, 2016.

[40] J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,“ *In IEEE conference on computer vision and pattern recognition*, pp. 248-–255, 2009.

[41] P. G. Mousouliotis, and L. P. Petrou, “CNN-Grinder: From Algorithmic to High-Level Synthesis Descriptions of CNNs for Low-end-low-cost FPGA SoCs,” *Microprocessors and Microsystems*, vol. 102990, 2020.

[42] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, et al., “A configurable cloud-scale DNN processor for real-time AI,“ *In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, 2018.

[43] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, et al., “Deep speech: Scaling up end-to-end speech recognition,“ *arXiv preprint arXiv:1412.5567*, 2014.

[44] C. Gao, D. Neil, E. Ceolini, S. C. Liu, and T. Delbruck, “DeltaRNN: A power-efficient recurrent neural network accelerator,“ *In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 21–30, 2018.

[45] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, 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)*, pp. 152–159, 2017.

[46] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, “Finn: A framework for fast, scalable binarized neural network inference,“ *In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, 2017.

[47] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, “Ternary neural networks for resource-efficient AI applications,“ *In international joint conference on neural networks (IJCNN)*, pp. 2547–2554, 2017.

[48] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, et al., “Think fast: a tensor streaming processor (TSP) for accelerating deep learning workloads,“ *In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 145–158, 2020.

[49] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, et al.,“SCNN: An accelerator for compressed-sparse convolutional neural networks,“ *In ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 27-–40, 2017.

[50] H. Jang, J. Kim, J. E. Jo, J. Lee, and J. Kim, "MnnFast: A fast and scalable system architecture for memoryaugmented neural networks," *In International Symposium on Computer Architecture (ISCA)*, pp. 250-–263, 2019.

[51] A. D. Lascorz, S. Sharify, I. Edo, D. M. Stuart, O. M. Awad, P. Judd, et al.,"ShapeShifter: enabling fine-grain data width adaptation in deep learning," *In International Symposium on Microarchitecture (MICRO)*, pp. 28-–41, 2019.

[52] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, 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*, pp. 75–84, 2017.

[53] S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang,"C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs," *In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 11–20, 2018.

[54] T. M. Le, W. M. Snelgrove, and S. Panchanathan, "SIMD processor arrays for image and video processing: a review," *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 3311, pp. 30–41, 1998.

[55] D. G. Elliott, M. Snelgrove, and M. Stumm, "Computational-RAM: a memory-SIMD hybrid and its applications to DSP," *IEEE Custom Integrated Circuits Conference*, pp. 30.6.1–30.6.4, Boston, 1992.

[56] C. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, et al., "Vector IRAM: A Media-oriented Vector Processor with Embedded DRAM," *12th Hot Chips Conference, Palo Alto, CA*, 2000.

[57] M. Gokhale, B. Holmes, and K. Iobst, "Processing in Memory: The Terasys Massively Parallel PIM Array," *IEEE Computer*, vol. 28, no. 3, pp. 23–31, 1995.

[58] P. M. Kogge, T. Sunaga, and E. A. E. Retter, "Combined DRAM and logic chip for massively parallel applications," *16th IEEE Conference on Advanced Research in VLSI, Raleigh, NC*, 1995.

[59] C . Lehmann, M. Viredaz, and F. Blayo, "A generic systolic array building block for neural networks with on-chip learning," *IEEE transactions on neural networks*, vol. 4, no. 3, pp. 400-407, 1993.

[60] J. Shen, H. Ren, Z. Zhang, J. Wu, W. Pan, and Z. Jiang, "A High-Performance Systolic Array Accelerator Dedicated for CNN," *In 2019 IEEE 19th International Conference on Communication Technology (ICCT)*, pp. 1200–1204, 2019.

[61] D. Andrews, C. Kancler, and B. Wealand, "An embedded real-time SIMD processor array for image processing, " *In Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems*, pp. 131–134, 1998.

[62] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, et al., "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs, " *In Proceedings of the 54th Annual Design Automation Conference*, pp. 1–6, 2017.

[63] C. Kyrkou and T. Theocharides, "SCoPE: Towards a systolic array for SVM object detection, " *IEEE Embedded Systems Letters*, vol. 1, no. 2, 46–49, 2009.

[64] D. Koch, C. Beckhoff, and G. G. Lemieux, "An efficient FPGA overlay for portable custom instruction set extensions," *In 23rd international conference on field programmable logic and applications*, pp. 1–8, 2013.

[65] S. McGettrick, K. Patel, and C. Bleakley, "High performance programmable FPGA overlay for digital signal processing," *In International Symposium on Applied Reconfigurable Computing*, pp. 375–384, 2011.

[66] L. B. D. Silva, R. Ferreira, M. Canesche, M. M. Menezes, M. D. Vieira, J. Penha, et al., "READY: A Fine-Grained Multithreading Overlay Framework for Modern CPU-FPGA Dataflow Applications," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–20, 2019.

[67] A. Brant and G. G. Lemieux, "ZUMA: An open FPGA overlay architecture," *In 2012 IEEE 20th international symposium on field-programmable custom computing machines*, pp. 93–96, 2012.

[68] A. Landy and G. Stitt, "Serial Arithmetic Strategies for Improving FPGA Throughput," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 3, pp. 1–25, 2017.

[69] A. J. Miller, U.S. Patent No. 6,438,570. Washington, DC: U.S. Patent and Trademark Office, 2002.

[70] A. Landy and G. Stitt, "Revisiting serial arithmetic: A performance and tradeoff analysis for parallel applications on modern FPGAs," *In 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 9–16, 2015.

[71] D. Okamoto, M. Kondo, T. Yokogawa, Y. Sejima, K. Arimoto, and Y. Sato, "A Serial Booth Multiplier Using Ring Oscillator," *In 2016 Fourth International Symposium on Computing and Networking (CANDAR)*, pp. 458–461, 2016.

[72] N. Kapre, "On bit-serial NoCs for FPGAs," *In 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 32–39, 2017.

[73] A. Rettberg, M. Zanella, C. Bobda, and T. Lehmann, "A fully self-timed bit-serial pipeline architecture for embedded systems," *In Design, Automation and Test in Europe Conference and Exhibition*, pp. 1130–1131, 2003.

[74] A. Rettberg, M. Zanella, T. Lehmann, and C. Bobda, "A new approach of a self-timed bit-serial synchronous pipeline architecture," *In 14th IEEE International Workshop on Rapid Systems Prototyping*, pp. 71–77, 2003.

[75] B. M. Li and P. H. Leong, "Serial and parallel FPGA-based variable block size motion estimation processors," *Journal of Signal Processing Systems*, vol. 51, no. 1, pp. 77–98, 2008.

[76] D. Walsh and P. Dudek, "A compact FPGA implementation of a bit-serial SIMD cellular processor array," *In Proceedings of the 13th International Workshop on Cellular Nanoscale Networks and their Applications, Turin*, pp. 1–6, 2012.

[77] M. P. Leong, O. Y. Cheung, K. H. Tsoi, and P. H. W. Leong, "A bit-serial implementation of the international data encryption algorithm IDEA," *In Proceedings Symposium on Field-Programmable Custom Computing Machines*, pp. 122–131, 2000.

[78] O. Y. Cheung, K. H. Tsoi, P. H. W. Leong, and M. P. Leong, "Tradeoffs in parallel and serial implementations of the international data encryption algorithm IDEA," *In International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 333–347, 2001.

[79] D. J. Moss, D. Boland, and P. H. Leong, "A Two-Speed, Radix-4, Serial–Parallel Multiplier," *Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 4, pp. 769–777, 2018.

[80] K. Oguri, Y. Shibata, and A. Nagoya, "Asynchronous bit-serial datapath for object-oriented reconfigurable architecture PCA," *In Asia-Pacific Conference on Advances in Computer Systems Architecture, Springer, Berlin, Heidelberg*, pp. 54–68, 2003.

[81] S. Dick, V. Gaudet, and H. Bai, "Bit-serial arithmetic: A novel approach to fuzzy hardware implementation," *In NAFIPS 2008-2008 Annual Meeting of the North American Fuzzy Information Processing Society*, pp. 1–6, 2008.

[82] A. D. Booth, "A signed binary multiplication technique. The Quarterly Journal of Mechanics and Applied Mathematics," vol. 4, no. 2, pp. 236-240, 1951.

[83] G. Csordás, B. Fehér, and T. Kovácsházy, "Application of bit-serial arithmetic units for FPGA implementation of convolutional neural networks," *In 2018 19th International Carpathian Control Conference (ICCC)*, pp. 322-327, 2018.

[84] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," *In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 764–775, 2018.

[85] O. Bilaniuk, S. Wagner, Y. Savaria, and J. P. David, "Bit-slicing fpga accelerator for quantized neural networks," *In 2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2019.

[86] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," *In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[87] Z. He, Z. Wang, and G. Alonso, "Bis-km: Enabling any-precision k-means on fpgas," *In Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 233–243, 2020.

[88] M. K. Shuvo, "Hardware Efficient Deep Neural Network Implementation on FPGA," *(Doctoral dissertation, Southern Illinois University at Carbondale)*, 2020.

[89] J. Skodzik, V. Altmann, B. Wagner, P. Danielis, and D. Timmermann, "A highly integrable fpga-based runtime-configurable multilayer perceptron," *In IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, pp. 429–436, 2013.

[90] N. B. Gaikwad, V. Tiwari, A. Keskar, and N. C. Shivaprakash, "Efficient FPGA implementation of multilayer perceptron for real-time human activity classification," *IEEE Access*, vol. 7, pp. 26696–26706, 2019.

[91] A. Jain, E. D. Pitchika, and S. Bharadwaj, "An exploration of fpga based multilayer perceptron using residue number system for space applications," *In 14th IEEE International Conference on Signal Processing (ICSP)*, pp. 1050–1055, 2018.

[92] P. Colangelo, O. Segal, A. Speicher, and M. Margala, "AutoML for Multilayer Perceptron and FPGA Co-design," *arXiv preprint arXiv:2009.06156*, 2020.

[93] F. Benrekia, M. Attari, and M. Bouhedda, "Gas sensors characterization and multilayer perceptron (MLP) hardware implementation for gas identification using a field programmable gate array (FPGA). Sensors", vol. 13, no. 3, pp. 2967–2985, 2013.

[94] A. N. Perez-Garcia, G. M. Tornez-Xavier, L. M. Flores-Nava, F. Gómez-Castañeda, and J. A. Moreno-Cadenas, "Multilayer perceptron network with integrated training algorithm in FPGA," *In 11th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*, pp. 1–6, 2014.

[95] A. Sanaullah, C. Yang, Y. Alexeev, K. Yoshii, andM. C. Herbordt, "Real-time data analysis for medical diagnosis using FPGA-accelerated neural networks," *BMC bioinformatics*, vol. 19, no. 18, pp. 19–31, 2018.

[96] S. Siddhartha, S. Wilton, D. Boland, B. Flower, P. Blackmore, and P. Leong, "Simultaneous inference and training using on-fpga weight perturbation techniques," *In International Conference on Field-Programmable Technology (FPT)*, pp. 306–309, 2018.

[97] A. Sanaullah, C. Yang, Y. Alexeev, K. Yoshii, and M. C. Herbordt, "Application aware tuning of reconfigurable multi-layer perceptron architectures," *In IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–9, 2018.

[98] M. Wess, P. S. Manoj, and A. Jantsch, "Neural network based ECG anomaly detection on FPGA and trade-off analysis," *In IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, 2017.

[99] P. R. Gankidi and J. Thangavelautham, "FPGA architecture for deep learning and its application to planetary robotics," *In IEEE Aerospace Conference*, pp. 1–9, 2017.

[100] K. Chen, L. Huang, M. Li, X. Zeng, and Y. Fan, "A compact and configurable long short-term memory neural network hardware architecture," *In 25th IEEE International Conference on Image Processing (ICIP)*, pp. 4168–4172, 2018.

[101] Y. Zheng, H. Yang, Z. Huang, T. Li, and Y. Jia, "A High Energy-Efficiency FPGA-Based LSTM Accelerator Architecture Design by Structured Pruning and Normalized Linear Quantization," *In International Conference on Field-Programmable Technology (ICFPT)*, pp. 271–274, 2019.

[102] Y. Zhang, C. Wang, L. Gong, Y. Lu, F. Sun, C. Xu, et. al, "A power-efficient accelerator based on FPGAs for LSTM network," *In IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 629–630, 2017.

[103] S. Wang, P. Lin, R. Hu, H. Wang, J. He, Q. Huang, and S. Chang, "Acceleration of LSTM with structured pruning method on FPGA," *IEEE Access*, vol. 7, pp. 62930–62937, 2019.

[104] J. C. Ferreira, and J. Fonseca, "An FPGA implementation of a long short-term memory neural network," *In International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–8, 2016.

[105] G. Maor, X. Zeng, Z. Wang, Y. and Hu, Y. "An FPGA Implementation of Stochastic Computing-Based LSTM," *In IEEE 37th International Conference on Computer Design (ICCD)*, pp. 38–46, 2019.

[106] M. S. Roodsari, M. A. Saber, and Z. Navabi, "DiBA: n-Dimensional Bitslice Architecture for LSTM Implementation," *In 23rd International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pp. 1–6, 2020.

[107] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, et. al, "Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity," *IIn Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 63–72, 2019.

[108] V. Rybalkin, A. Pappalardo, M. M. Ghaffar, G. Gambardella, N. Wehn, and M. Blott, "FINN-L: Library extensions and design trade-off analysis for variable precision LSTM networks on FPGAs," *In 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 889–897, 2018.

[109] C. Heelan, A. V. Nurmikko, and W. Truccolo, "FPGA implementation of deep-learning recurrent neural networks with sub-millisecond real-time latency for BCI-decoding of large-scale neural sensors (104 nodes)," *In 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pp. 1070–1073, 2018.

[110] C. L. Li, Y. J. Huang, Y. J. Cai, J. Han, and X. Y. Zeng, "FPGA implementation of LSTM based on automatic speech recognition," *In 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pp. 1–3, 2018.

[111] V. Rybalkin, N. Wehn, M. R. Yousefi, and D. Stricker, "Hardware architecture of bidirectional long short-term memory neural network for optical character recognition," 2017.

[112] Y. Zhang, C. Wang, L. Gong, Y. Lu, F. Sun, C. Xu, et. al, "Implementation and optimization of the accelerator based on fpga hardware for lstm network," *In IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, pp. 614–621, 2017.

[113] Y. Yamauchi, K. Musha, and H. Amano, "Implementing a large application (LSTM) on the multi-FPGA system: Flow-in-Cloud," *In IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, pp. 1–3, 2019.

[114] E. Bank-Tavakoli, S. A. Ghasemzadeh, M. Kamal, A. Afzali-Kusha, and M. Pedram, "Polar: A pipelined/overlapped fpga-based lstm accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 3, pp. 838–842, 2019.

[115] T. Kaneko, H. Momose, and T. Asai, "An FPGA Accelerator for Embedded Microcontrollers Implementing a Ternarized Backpropagation Algorithm," *In International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–8, 2019.

[116] R. Gadea, J. Cerdá, F. Ballester, and A. Mocholí, "Artificial neural network implementation on a single FPGA of a pipelined on-line backpropagation," *In Proceedings of the 13th international symposium on System synthesis*, pp. 225–230, 2000.

[117] H. M. Vo, "Implementing the on-chip backpropagation learning algorithm on FPGA architecture," *In International Conference on System Science and Engineering (ICSSE)*, pp. 538–541, 2017.

[118] Q. Liu, J. Liu, R. Sang, J. Li, T. Zhang, and Q. Zhang, "Fast neural network training on FPGA using quasi-Newton optimization method," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 8, pp. 1575–1579, 2018.

[119] X. Xie, X. Li, D. Li, and L. Xu, "Real-Time In-Situ Laser Ranging via Back Propagation Neural Network on FPGA," *IEEE Sensors Journal*, vol. 21, no. 4, pp. 4664–4673, 2020.

[120] B. Girau and A. Tisserand, "On-line arithmetic-based reprogrammable hardware implementation of multilayer perceptron back-propagation," *In Proceedings of Fifth International Conference on Microelectronics for Neural Networks*, pp. 168–175, 1996.

[121] A. W. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study," *IEEE transactions on neural networks*, vol. 18, no. 1, pp. 240–252, 2007.

[122] M. Imani, Z. Zou, S. Bosch, S. A. Rao, S. Salamat, V. Kumar, and et. al, "Revisiting hyperdimensional learning for fpga and low-power architectures," *In 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 221–234, 2021.

[123] F. Ortega-Zamorano, J. M. Jerez, D. U. Munoz, R. M. Luque-Baena, and L. Franco, "Efficient implementation of the backpropagation algorithm in FPGAs and microcontrollers," *IEEE transactions on neural networks and learning systems*, vol. 27, no. 9, pp. 1840–1850, 2015.

[124] J. Liu and Q. Liu, "Speed and resource optimization of BFGS quasi-Newton implementation on FPGA using inexact line search method for neural network training," *In International Conference on Field-Programmable Technology (FPT)*, pp. 362–365, 2018.

[125] Q. Li, S. Fleming, D. Thomas, and P. Cheung, "Accelerating Top-k ListNet Training for Ranking Using FPGA," *In International Conference on Field-Programmable Technology (FPT)*, pp. 242–245, 2018.

[126] High Rate SHM Working Group, https://github.com/High-Rate-SHM-Working-Group/Dataset-4-Univariate-signal-withnon-stationarity, Dataset-4 univariate signal with nonstationarity.

[127] S. Hochreiter, J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, 1997.