ReMoDeL-FPGA: Reconfigurable Memory-centric Array Processor Architecture for Deep-Learning Acceleration on FPGA

MD Arafat Kabir
University of Arkansas, Fayetteville

ReMoDeL-FPGA: Reconfigurable Memory-centric Array Processor Architecture for Deep-Learning Acceleration on FPGA

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Engineering

by

MD Arafat Kabir
Bangladesh University of Engineering and Technology
Bachelor of Science in Electrical and Electronic Engineering, 2017
University of Arkansas
Master of Science in Computer Engineering, 2021

August 2024
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council.

David Andrews, Ph.D.
Dissertation Director

Jia Di, Ph.D.
Committee Member

Alexander Nelson, Ph.D.
Committee Member

Zhong Chen, Ph.D.
Committee Member

Abstract

Deep-Learning has become a dominant computing paradigm across a broad range of application domains. Different architectures of Deep-Networks like CNN, MLP, and RNN have emerged as the prominent machine-learning approaches for today's application domains. These architectures are heavily data-dependent, requiring frequent access to memory. As a result, these applications suffer the most from the memory bottleneck of the von Neumann architectures. There is an imminent need for memory-centric architectures for deep-learning and big-data analytic applications that are memory intensive. Modern Field Programmable Gate Arrays (FPGAs) are ideal programmable substrates for creating customized Processor in/near Memory (PIM) accelerators. Modern FPGAs contain 100s of Mbits of dual-ported SRAM in the form of disaggregated, configurable Block RAMs (BRAMs). These BRAMs contain TB/s of available internal bandwidth.

Unfortunately, developing FPGA-based accelerators for deep learning is not a simple task and demands the utilization of specialized tools provided by the FPGA vendors. It requires expertise in low-level hardware microarchitecture design. These are often not available to most researchers in the field of deep learning. Even with the ongoing improvements in High-Level Synthesis (HLS) tools, the requirement for hardware-specific design knowledge cannot be completely eliminated.

This research developed a new reconfigurable memory-centric architecture and design approach that opens the advantages of FPGAs and Processor-in-Memory architecture to memory-intensive applications. Due to its high-performance and scalable memory-centric design, this architecture can deliver the highest speed and the lowest latency achievable from an FPGA overcoming the memory bottleneck.

Acknowledgements

First, I would like to express my deepest gratitude to my dissertation advisor, Dr. David Andrews, who is undoubtedly the wisest person I have ever met. His guidance has been instrumental not only in my academic journey but also in shaping my understanding of life. Through our numerous discussions, I have gained invaluable insights, and by observing his interactions with current and former students, colleagues, and acquaintances, I have learned the importance of empathy, integrity, and perseverance. His unwavering support and profound wisdom have inspired me to strive for excellence in all aspects of my life. Thank you for being an extraordinary mentor and a true role model.

I would like to thank Dr. Zhong Chen, Dr. Jia Di, and Dr. Alexander Nelson for taking time out of their schedules to participate in the committee for this dissertation. I am particularly grateful to Dr. Nelson for his understanding and support during my challenging times as a graduate student. I would like to thank Dr. Di for teaching and helping me in the completion of the required courses for this degree. I would like to thank Dr. Chen for always being available whenever I needed his support. I am grateful for all the help and cooperation I have received from them when I needed them most.

I want to express my gratitude to the National Science Foundation (NSF) for supporting the project that led to this dissertation. NSF provided the funding and support that allowed me to pursue my graduate degree. I would like to thank my colleagues, Tendayi Kamucheka and Ehsan Kabir, for supporting me and working with me throughout the time of this research.

Finally, I would like to thank my family and friends who have all supported me in every step of the way to achieving this goal. It would not have been possible for me to finish this degree without all of them. With far too many names to list, I would like to tell them how thankful I am for their care, support, and motivation during this journey.

Dedication

To my mother, who unconditionally supported me to pursue my dreams

To my father, who always believed in me

To my brother, who showed me how not to lose hope in difficult times

To my wife, who has been by my side supporting me whenever I stumbled

This dissertation and all of my achievements are the results of the care and support I have received from you all. Without your continued belief in me to succeed, I would not be where I am today.

I LOVE YOU ALL

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep Neural Networks (DNNs) are widely used in Artificial Intelligence applications such as computer vision, voice recognition, autonomous driving, and robotics. With the help of advanced digital technologies and data handling infrastructure, DNNs have become an increasingly popular choice for solving complex real-world problems. In some cases, the performance and accuracy of a DNN can exceed that of human intelligence. However, these networks are computationally demanding, requiring a large number of resources to perform data-intensive calculations. General-purpose architectures like CPUs are not well-suited to handle such data and computationally intensive algorithms. As a result, there has been significant interest and effort invested in developing specialized architectures for different hardware platforms including Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), Application Specific Integrated Circuits (ASICs), and Coarse-Grained Reconfigurable Arrays (CGRAs) to enable more effective implementation of these computationally intensive algorithms.

In the past decade, several FPGA-based custom accelerator designs have been proposed for deep-learning applications [4–22]. Creating a custom accelerator that is specifically tailored to a particular application is an ideal solution as it matches resources to the problem at hand and has a relatively simple design complexity. The accelerator architecture can be fine-tuned for the particular application taking advantage of the unique features of the target device. Custom accelerators achieve the shortest latency and highest performance with the lowest power consumption. However, they are not reusable for other applications and may lose performance when synthesized for a different device. Moreover, a custom accelerator takes a very long time to design and is not portable between FPGA families.

To solve these portability issues, several domain-specific overlay architectures have been proposed for deep-learning [23–31]. They reduce design complexity by abstracting the underlying hardware and providing a simpler interface to programming it for a particular application. They improve the programmer's productivity and design speed by eliminating the synthesis step. However, it comes at the cost of reduced performance, higher latency, worse FPGA resource utilization, and higher power.

An interesting design approach has evolved in the form of High-Leve Synthesis (HLS), which tries to bridge the gap between software programming and hardware accelerator design. Several HLS-based deep-learning accelerators have been proposed [32–37] in the past few years. However, there is a considerable difference in the achievable frequency between an HLS design and a manually-crafted RTL design. Interconnect delays are more challenging to predict and optimize for HLS tools, which often results in interconnect delays becoming the frequency bottleneck for HLS-based designs. Furthermore, modern FPGAs are increasingly heterogeneous and can span multiple dies, resulting in longer routing paths that can become the limiting factor for the maximum frequency of large designs. As a result, timing issues are much worse in such scenarios.

An outcome of this dissertation is a reconfigurable memory-centric array processor architecture for deep-learning applications on FPGA that breaks the von Neumann memory bottleneck and provides the maximum speed the target device can support. Its reconfigurable design and abstract programming interfaces offer the portability and programmer productivity of overlays and HLS. At the same time, it provides the highest performance and maximum utilization of the device's internal memory bandwidth, making it competitive with custom accelerators.

1.1 Why Array Processors?

Array processors are the most suitable candidates for deep-learning accelerators for several reasons including their massive parallelism and distributed memory architecture. Deep-learning applications are inherently parallel. The most common operations like multiply-accumulate (MAC), convolution, and pooling employ the same computation on different data. This computing model resembles single-instruction multiple-data (SIMD) architecture. In the ideal case, we want as many processing elements as the number of parallel instances of such operations. This is exactly what array processors offer. Array processors have SIMD architecture with thousands of processing elements that can carry out those operations in parallel, taking full advantage of the inherent parallelism of the deep-learning models.

On the other hand, some deep-learning applications are computation-bound while others are memory-bound. Convolutional Neural Networks (CNNs), Multilayer Perceptrons (MLPs), and Recurrent Neural Networks (RNNs) have emerged as the dominant machine learning approaches for today's application domains. Each of the three networks has different computation to communication requirements. These requirements are analytically quantified by an operational intensity computed as the number of multiply-accumulate (MAC) operations performed per weight. CNNs exhibit high operational intensities where end-to-end inference latencies are dominated by arithmetic compute times. Conversely, MLPs and RNNs exhibit low operational intensities where the end-to-end inference latencies are dominated by bus bandwidth and memory swapping times. Processor in/near memory (PIM) architectures are making a resurgence to address these types of networks. PIM architectures integrate bit-serial processors within memory. This eliminates the sequential bottleneck between the processor and memory resources in traditional von Neumann architectures. PIM systems offer a theoretical peak performance limited only by the memory bandwidth. These PIM blocks when used as the processing elements in array processors provide very high memory bandwidth along with massive parallelism.

1.2 Why FPGAs?

An argument in favor of not using FPGAs is that they are much slower than application-specific ICs (ASICs). A TSMC 28nm ASIC can run at 3 GHz, while the Virtex 7 FPGA designed in the same technology node can run as fast as 600 MHz, $5\times$ slower than ASIC. However, ASICs have a long turnaround time. If we look at Google's Tensor Processing Unit (TPU) as an example, it took a team of experts in the architecture world 15 months to design it with almost unlimited resources available at Google. This time frame can easily become 2 – 4 years for an average engineering team. On the other hand, an FPGA accelerator design can be completed in a few months.

Deep-learning models are fast-evolving. In 2014, the VGGNet architecture achieved state-of-the-art results on the ImageNet classification task with a top-5 error rate of 7.3%. In 2016, the ResNet architecture achieved even better performance on the ImageNet classification task, with a top-5 error rate of 3.57%. Thus, in just two years, the ResNet architecture was able to outperform the VGGNet architecture by a significant margin on the ImageNet classification task. This is just one example of how rapidly deep-learning architectures can evolve and improve over a short period of time. ASIC design cycles cannot keep up with this fast growth of deep learning applications. Thus, despite being significantly slower, FPGAs are the most suitable candidates for deep-learning accelerators compared to ASIC.

Moreover, FPGAs are an ideal reconfigurable platform for memory-centric Array Processor implementation. If we look at the layout of a Virtex-7 or an UltraScale FPGA, they have distributed BRAMs all over the reconfigurable fabric. It is not difficult to see that, these BRAMs can be utilized to build PIM blocks and then connect them together to build an Array Processor. Recent proposals have been made to modify the BRAM tiles to convert them to PIM blocks. However, these are not currently available on mainstream devices.

1.3 Why Overlays?

The process of developing applications for FPGAs is vastly different from writing software programs, with a higher degree of complexity and intricacy involved. It requires intimate knowledge of the underlying FPGA architecture as well as experience in low-level hardware microarchitecture design. In addition, software programmers may face difficulties in working with complicated vendor-specific implementation tools, which are designed specifically for their FPGA development. The usage of these tools requires an understanding of hardware design flows. Moreover, the synthesis step involved in programming for FPGAs can be time-consuming, leading to slower turnaround times. Therefore, programmers who are accustomed to the rapid iteration cycles of software development may find FPGA development to be a more arduous and challenging process.

An FPGA overlay is a virtual reconfigurable architecture that overlays on top of the physical FPGA configurable fabric. Overlays provide a level of abstraction to the underlying FPGA hardware, allowing for greater reuse, versatility, and programmer accessibility. Applications developed for a particular overlay can be easily transferred to other hardware systems that use the same overlay, enabling efficient code reuse. The development cycle for overlays is similar to that of software development, with quick turnaround times and a greater degree of flexibility. Additionally, overlays can be used with most off-the-shelf FPGAs available on the market, providing greater accessibility and availability for developers. Furthermore, applications developed with overlays automatically benefit from performance improvements when running on new generations of FPGAs, ensuring a greater degree of scalability and future-proofing. All these reasons strongly motivate the development of a reconfigurable overlay to make FPGAs more accessible as deep-learning accelerators.

1.4 Why Application-Specific Customization?

This is probably the easiest question to answer. A design that is good for everyone is best for none. Different deep-learning models have different computation and data-movement patterns that necessitate different types of architectural support. If we look at Google's data-center report on TPU [38], we can see,

- Array active cycles are high for CNN but very low for MLP and LSTM
- Weight stall cycles for MLP and LSTM are high but for CNN very low
- Weight shift cycles for MLP and LSTM are significant, but CNN is insignificant
- Non-matrix cycles of all of them are comparable

These results suggest that if we want to improve the performance of CNNs, we need to focus on the computation and control part of the accelerator. We should invest more resources and optimization efforts to improve the clock frequency and throughput of the processing elements and the control circuitry and can relax some constraints on the data network. On the other hand, for MLP and LSTM, we need to focus more on designing a better data network to decrease or hide the weight stalls as much as possible. It is okay if we relax some constraints on the computation circuit and relocate the resources for a better network design.

Another argument in favor of application-specific customization is the constraints imposed by the application environment. The same deep-learning model can have different sets of constraints based on the runtime environment. Let's take AlexNet for example, which is a CNN designed for image classification. If AlexNet will be used by a user sitting in front of a desktop computer to process some images on its hard drive, it is probably a better idea to optimize the accelerator for throughput and not worry about latency and power too much. On the other hand, if it will be used by a battery-powered space probe as part of its navigation system it probably needs to have shorter inference latency and be power efficient. The point is, the same deep-learning model needs to meet different constraints for different application environments and so, the accelerator needs to be customized to meet those constraints.

Moreover, different FPGAs have different logic resources, at least they vary in the distribution of those resources; some have more memory and less logic than others, some have faster DSPs, while some of the embedded FPGAs don't even have DSP blocks. So, the accelerator overlay need to be customizable to take advantage of the unique resources available in the target device as well as adjust its resource budgets for each component of the architecture based on the deep-learning model, application goals, and the resource distribution of the target device.

1.5 Thesis Statement

This dissertation explored the question of the feasibility of creating a Reconfigurable Memory-Centric Array Processor Overlay architecture that can approach an optimal solution for FPGA-based deep-learning accelerators. This statement has been evaluated by developing and studying a reconfigurable memory-centric Array Processor Overlay architecture and design approach. To be broad, this work broke down and investigated the question for each microarchitecture subsystem of an Array Processor to understand how each subsystem impacts performance at the system level across the major categories of deep-learning applications. This study resulted in the formulation of a theoretical upper limit for BRAM-LUT-based PIM array architectures in the form of a set of ideal design standards. These standards are to be used as an aspirational set of design objectives as well as guiding principles for making near-optimal architectural design choices. To be specific, the contribution of this dissertation is a Reconfigurable memory-centric array processor overlay architecture that,

- Establishes an ideal design standard for high-performance PIM designs,
- Runs as fast as the maximum frequency of the BRAM,
- Scales its peak-performance linearly with the BRAM capacity of the target device,
- Offers a balanced network design for reduction operations across the array,
- Offers full customization access at both software and hardware levels,
- Outperforms most of the existing PIM accelerators and many custom accelerators in terms of application latency.

Chapter 2

Related Work

Over the past few decades, several FPGA-based accelerators have been developed for deep-learning applications. This chapter briefly discusses some of the more recent and relevant research work on FPGA-based accelerator design for deep-learning.

2.1 FPGA-Based Custom Accelerators

Custom accelerators designed for a particular application can provide the best performance and power at a very small cost. Moreover, they can be fine-tuned for the target platform to take advantage of the platform-specific features relevant to the application.

Han et al. [7] proposed Efficient speech recognition engine (ESE) to accelerate speech recognition using LSTM on FPGA. The design presented in this study utilizes a load-balanced sensing pruning technique to compress the LSTM model. To implement the LSTM algorithm for speech recognition, the proposed accelerator employs the Kaldi framework. Running at 200 MHz on a Xilinx XCKU060 FPGA, the proposed design achieves a performance of 282 GOPS. Additionally, the study showcases the implementation of speech recognition algorithms using FPGA-based accelerators, which has also been discussed in previous studies [4, 11, 16, 20].

Wang et al. [17] proposed YOLOv3 to accelerate object detection using FPGA. The YOLOv3 (You Only Look Once, Version 3) is an object detection algorithm that detects specific objects in images or videos in real time. A new accelerator based on the ARM with FPGA architecture has been proposed. The experimental results demonstrate that this FPGA-based YOLOv3 accelerator consumes less energy and achieves higher throughput than its GPU counterpart. Furthermore, it is compatible with various frameworks, including TensorFlow, Caffe, and PyTorch. The implementation of the proposed accelerator is performed on Xilinx ZCU104 with a frequency of 300 MHz. This approach is also used in several other works to implement object detection algorithms [6, 12, 13].

Hamza et al. [28] proposed NPE to efficiently implement Natural Language Processing (NLP) models on FPGA. The NPE offers a unified platform to handle non-linear functions of any complexity, with programmability similar to that of software. Compared to CPUs and GPUs, the NPE consumes 4 times and 6 times less power respectively. It has been implemented on the Xilinx Zynq Z-7100 FPGA, operating at a frequency of 200 MHz.

Lian et al. [24] have presented a CNN accelerator based on block-floating-point (BFP) arithmetic to support DNN inference. The accelerator includes Processing Element Array (PEA), an on-chip buffer, and external memory. The proposed accelerator used 8-bit and 16-bit formats to represent the feature maps and model parameters. This approach decreased off-chip bandwidth and memory, while only slightly affecting accuracy in comparison to the 32-bit floating-point approach. The accelerator design was performed with the Vivado HLS tool, and the proposed BFP arithmetic is conducted with the Caffe [23] scheme. The Xilinx VC709 evaluation board is utilized to implement the accelerator, which runs at 200 MHz and achieves a throughput of 760.83 GOP/s.

Xiao et al. [18] proposed a DNN accelerator designed for sparse and compressed DNN models. The accelerator includes a PE array, RLC encoder, controller, and on-chip buffers. The weights and non-linear activation functions are stored in RLC compressed form in off-chip DRAM memory. The PE array contains 64 PEs and performs the MAC operations of the fully connected layer. A new dataflow is proposed to reuse input activations across fully connected layers to exploit parallelism and maximize the utilization of PEs. The proposed architecture achieved a performance of 1.34 GOP/s on the Xilinx Virtex-7 FPGA platform.

Ahmed et al. [19] introduced an FPGA-based low-power CNN accelerator for GoogLeNet CNN. The proposed accelerator implements quantization and weight pruning techniques to decrease the memory size, enabling pipelining and processing layer by layer. The on-chip memory stores activations and weights, replacing multiplication with shifting operations, and avoiding DSP units. The LP-CNN accelerator provides a significant improvement in power efficiency over Intel Core-i7 and NVidia GTX 1080Ti, achieving 49.5 and 7.8 times power improvement, respectively. The LP-CNN accelerator has been implemented on the Virtex-7 FPGA at 200 MHz.

In [26], a low-power FPGA-based accelerator designed for LeNet CNNs was introduced. The accelerator utilizes 8-bit, 16-bit, and 32-bit fixed point representations for the weights, activations, and biases, respectively. It supports pipelining and implements LeNet using minimal resources to maintain throughput. The design process involves converting C++ code to RTL implementation using the Xilinx Vitis HLS tool. The accelerator is implemented on the Nexys DDR 4 FPGA evaluation board and is capable of processing 14,000 images per second while consuming only 628 mW of power.

An FPGA-based dynamically reconfigurable architecture was presented in [27] that uses Dynamic Partial Reconfiguration to switch between different types of neural network architectures without affecting precision or throughput. The proposed accelerator includes a PE array and configurable switches controlled by a hard/soft processor. Each PE can implement various functionalities with the same hardware, and communicate with other PEs, CPUs, or I/O ports via configurable switches. The proposed accelerator was implemented on the Xilinx Zynq 7020 FPGA board and the C++ code was converted to RTL implementation using the Xilinx Vitis HLS tool.

Gowda et al. [30] proposed an FPGA-based reconfigurable CNN accelerator that contains configuration registers to reconfigure the architecture according to the instructions stored in the DDR. The PE arrays perform convolution operations while the special function buffer performs pooling, batch normalization, and activation functions. The proposed accelerator uses sparse optimization of weight and convolutional optimizations to reduce the sizes of weights and feature maps, respectively. It uses 16-bit, 8-bit, and 4-bit fixed point formats to represent the feature maps, CONV layer weights, and FC layer weights, respectively. The accelerator was implemented on the Xilinx Zynq 7020 FPGA board using the Vivado HLS toolchain.

Gao et al. [39] proposed a GRU-RNN accelerator architecture called DeltaRNN (DRNN). Its implementation was based on the Delta Network algorithm that skips dispensable computations during network inference by exploiting the temporal dependency in RNN inputs and activations. An implementation on a Xilinx Zynq-7100 FPGA of a single-layer RNN of 256 Gated Recurrent Unit (GRU) neurons showed that the DRNN achieved 1.2 TOp/s effective throughput and 164 GOp/s/W power efficiency. The delta update achieved 5.7$\times$ speedup compared to a conventional RNN update because of the sparsity created by the DN algorithm and the zero-skipping ability of DRNN.

Later, they proposed Spartus [40] improving upon DeltaRNN, that exploits the spatio-temporal sparsity to achieve ultralow latency inference. In Spartus, the spatial sparsity was induced using a column-balanced targeted dropout (CBTD) structured pruning method. The pruned networks running on Spartus hardware achieved weight sparsity levels of up to 96% and 94% with negligible accuracy loss on the TIMIT and the Librispeech datasets. Exploiting spatio-temporal sparsity, Spartus achieved per-sample latency of 1us for a single DeltaLSTM layer of 1024 neurons. It achieved 46$\times$ speedup for the LSTM network using the TIMIT dataset.

Several FPGA-based deep-learning accelerators have also been developed for medical purposes. A CNN accelerator was proposed by Serkan et al. [14] for the classification of malaria disease cells using FPGA. Xiong et al. [22] created a CNN accelerator using FPGA to enhance the automatic segmentation of 3D brain tumors. FPGA-based accelerators are also used to implement a variety of other applications such as autonomous driving [8, 10], image classification [5, 21], fraud detection [9], cancer detection [15], and more.

All these accelerators are application-specific. Their implementation depends either on the target device or the particular deep-learning model. These are neither reconfigurable nor memory-centric designs. Thus, this dissertation has no overlap with these works. However, some of these designs were used as comparison benchmarks to validate the performance of the accelerator developed in this dissertation.

2.2 Automated Accelerator Generation Frameworks

For rapid prototyping and to automate the process of deep learning accelerator generation, several automated frameworks have been proposed. Using these frameworks, it is possible to describe the accelerator in a high-level description language. Several approaches to high-level synthesis (HLS) of the accelerators have also been proposed. Some frameworks attempt to generate the accelerator directly from the software description of the deep-learning model.

Guan et al. [41] proposed FP-DNN (Field Programmable DNN), an end-to-end framework that automatically generates hardware implementations of deep neural networks described in TensorFlow for FPGA boards using RTL-HLS hybrid templates. The proposed framework performs model inference of DNNs using a high-performance computation engine and optimized communication strategies. FP-DNN was tested on several types of DNNs, including CNNs, LSTM-RNNs, and Residual Nets, and experimental results demonstrated the framework's flexibility and high performance. Results showed that using fixed16 data precision, FP-DNN outperformed CPU implementations by 1.9× to 3.06× in terms of speed. However, FP-DNN did not perform as well as GPU implementations. When it came to energy efficiency, FP-DNN always outperformed CPU implementations regardless of the model and precision used. In fact, FP-DNN could even surpass GPU implementations when using fixed16 data precision.

Wei et al. [42] propose a systolic array architecture, which is designed to increase the speed of convolutional neural networks on FPGAs. They use modeling techniques and design strategies to solve problems related to mapping, shape selection, and data reuse. The authors also developed a framework that automates the mapping process. Results show that their design can achieve up to 1171 GOPSon Intel's Arria 10 device.

Abdelfattah et al. [43] propose an overlay for deep neural network inference with a minimal overhead of about 1%. They use a lightweight VLIW network for control and reprogramming logic and a domain-specific graph compiler to compile deep learning languages such as Caffe and TensorFlow for their overlay. Their graph compiler optimizes the software architecture to improve the performance of CNNs and RNNs significantly. They report a 3× improvement on

ResNet-101 and a 12× improvement on LSTM cells compared to naive implementations.

Chi et al. [44] proposes SODA, an automated framework for implementing Stencil algorithms on FPGAs. SODA minimizes on-chip buffer size, provides flexible and scalable parallelism, and generates efficient high-frequency dataflow implementation. The automation framework reduces the difficulty of programming FPGAs efficiently for stencil algorithms. The SODA design-space exploration framework searches for the performance-optimized configuration with accurate models for resource utilization and onboard execution throughput. Experimental results show up to 3.28× speedup over 24-thread CPU, and the fully automated framework achieves better performance compared to manually designed state-of-the-art FPGA accelerators.

Cong et al. [45] proposed PolySA, an automated compilation framework that uses a polyhedral-based transformation to map algorithms to systolic array architecture. It enables efficient selection of scattering functions through the mapping process and generates off-the-shelf design IPs. PolySA can be used for matrix multiplication and convolutional neural networks, and it can identify all different design alternatives. It generates systolic array designs with comparable performance to well-tuned manual designs, and the entire compilation flow finishes within an hour. They also studied four common communication and computation patterns (scatter, gather, broadcast, and reduce) that can result in degraded frequency and long interconnects in scaled-out designs in High-Level Synthesis (HLS). To address this issue, the authors propose the Latte [46] microarchitecture, which uses pipeline transfer controllers in these four patterns to reduce wire delay and improve frequency. They also implement an automated framework to enable easy implementation of the Latte microarchitecture in HLS designs, which improves frequency from 120 MHz to 181 MHz on average with only a small overhead.

Yu et al. [47] proposed S2FA, an automated framework that can compile Spark application kernels to FPGA accelerators and integrate them into Blaze runtime. S2FA supports object-oriented constructs in bytecode-to-C code generation and uses a parallel learning-based design space exploration to optimize the accelerator performance. The framework generates FPGA kernels that provide up to 1225.2× and 49.9× speedup for string processing and machine learning applications, respectively, compared to the equivalent Scala implementations from which they are automatically generated.

Zhang et al. [48] proposed DNNBuilder, which delivers high performance and power efficiency for building DNN hardware accelerators on FPGAs, achieving the highest throughput performance peaking at 4218 GOPS (KU115) and 526 GOPS (ZC706) compared to existing FPGA/embedded FPGA-based solutions. It also achieved higher efficiency (up to 4.35x) than GPU-based solutions. The tool uses a fine-grained layer-based pipeline architecture and a column-based cache scheme for higher throughput, lower pipeline latency, and smaller on-chip memory consumption. It has a flexible process engine that provides optimal implementations of diversified DNN layers and allows adjusting parallelism factors to fit resource allocation guidelines.

Zhang et al. [49] proposed a uniformed convolutional matrix-multiplication representation for both convolutional and fully-connected layers, and optimize the accelerator microarchitecture based on a revised roofline model. They designed an automation flow to directly compile high-level network definitions to the final FPGA accelerator. Caffeine was integrated into the deep learning framework Caffe and achieved a peak performance of 1460 giga fixed point operations per second on a medium-sized FPGA board. It also achieved 29× and 150× performance and energy gains over Caffe on a 12-core Xeon server and 5.7× better energy efficiency over the GPU implementation.

Chen et al. [50] proposed a ternary hardware accelerator called T-DLA, and a framework for ternary neural network training. The T-DLA is a specialized hardware unit designed to accelerate TNNs, while the training framework can compress DNN parameters down to two bits with minimal accuracy loss. The authors report that their training framework can compress the DNN up to 14.14× while maintaining similar accuracy compared to the floating-point version. The T-DLA can deliver up to 0.4 TOPS with a power consumption of 2.576W, which is 873.6× and 5.1× more energy-efficient (fps/W) than the Xeon E5-2630 CPU and Nvidia 1080 Ti GPU on ImageNet with the Resnet-18 model.

Li et al. [51] proposed HeteroHalide, a system for compiling Halide programs to efficient FPGA accelerators, using the algorithm and scheduling information specified in a Halide program. Halide is a domain-specific language for image processing that allows programmers to search for optimized mappings targeting CPU and GPU. Heterohalid simplifies the flow for Halide programs to be applicable to FPGAs, requiring only moderate modifications on the scheduling part. HeteroCL is used as the intermediate representation of HeteroHalide, a heterogeneous programming infrastructure that supports multiple implementation backends. HeteroHalide generates efficient accelerators by choosing different backends according to the application. The performance evaluation shows that HeteroHalide achieves 4.15× speedup on average over 28 CPU cores and 2   4× throughput improvement compared with the existing Halide-HLS compiler. The proposed system provides a way to easily map Halide programs to efficient FPGA accelerators, which was previously a challenge.

Wang et al. [52] proposed AutoSA, a complete framework for compiling systolic arrays on FPGA. AutoSA utilizes the polyhedral framework and includes a series of optimizations across various dimensions to increase efficiency. It performs extensive design space exploration to identify high-performance designs. AutoSA's effectiveness has been proven across several applications where it has achieved high performance in a short period. For instance, in matrix multiplication, AutoSA achieves 934 GFLOPs, 3.41 TOPs, and 6.95 TOPs in floating-point, 16-bit, and 8-bit integer data types on the Xilinx Alveo U250.

Basalama et al. [53] proposed FlexCNN, an architecture for delivering high computation efficiency in convolutional neural networks (CNNs). It uses dynamic tiling, layer fusion, and data layout optimizations to achieve this. The architecture includes a versatile systolic array that can process normal, transposed, and dilated convolutions efficiently. It uses a fully pipelined software-hardware integration that reduces software overheads. With an automated compilation flow, FlexCNN takes an ONNX representation of a CNN, performs a design space exploration, and generates an FPGA accelerator. FlexCNN achieves 2.3× performance improvement and up to 15.98× and 13.42× speedups for transposed and dilated convolutions, respectively, with a 6%

average area overhead.

Liu et al. [54] proposed OverGen, a hardware generation framework targeting a highly customizable overlay. OverGen generated designs are as good as the fixed-function HLS-based systems. Even without tuning the kernel, OverGen provides 1.2 times better overall performance compared to AutoDSE, an HLS-based design exploration framework.

Zhuang et al. [55] proposed the CHARM framework, which allows for the composition of multiple MM (matrix multiplication) accelerator architectures that work concurrently on different layers of a single application. It uses analytical models for design space exploration to determine accelerator partitions and layer scheduling, and can automatically generate code for onboard design verification. CHARM was tested on four deep-learning applications on the AMD/Xilinx Versal ACAP VCK190 evaluation board. Results showed that CHARM achieved 1.46 TFLOPs, 1.61 TFLOPs, 1.74 TFLOPs, and 2.94 TFLOPs of inference throughput for BERT, ViT, NCF, and MLP, respectively. Compared to one monolithic accelerator, CHARM obtained throughput gains of 5.29x, 32.51x, 1.00x, and 1.00× for BERT, ViT, NCF, and MLP, respectively.

Due to the success of HLS [32] as a high-level interface to accelerator generation, several HLS-based automated frameworks have been proposed. Choi et al. [33] proposed the TARO framework, which automatically applies the free-running optimization on HLS-based streaming applications. Sohrabizadeh et al. [34] proposed AutoDSE that leverages a bottleneck-guided coordinate optimizer to systematically find a better design point. Ye et al. [35] proposed ScaleHLS, a scalable and customizable HLS framework, on top of a multi-level compiler infrastructure called MLIR. Jun et al. [36] proposed AutoScaleDSE, a design space exploration engine, which outperformed ScaleHLS by a maximum of 59X. Later, Ye et al. [37] proposed a new HLS framework combining ScaleHLS, AutoScaleDSE, and PyTransform, which promises to deliver a scalable and extensible solution to automate the process of designing domain-specific accelerators.

Though these frameworks automate the process of deep-learning accelerator generation, none of these targets memory-centric architectures. The placement and routing problems are NP-Hard, making it very difficult to get the highest performance using these automated approaches. As a result, the accelerators generated by these frameworks run significantly slower than application-specific custom accelerators. To quote Dr. Jason Cong, a leading expert in HLS and automated deep-learning accelerator generation, “There remains a significant gap in the achievable frequency between an HLS design and a hand-crafted RTL one.” [32] Thus, this research work does not overlap with these approaches. On the contrary, the memory-centric overlay architecture can act as the backend of these frameworks. For example, after all the polyhedral optimizations, these frameworks can generate a configuration for the overlay architecture targetting a particular device.

2.3 Memory-Centric Accelerators

In the past decade, researchers have been developing deep-learning accelerators for inference and training on various memory platforms. These platforms include traditional memory platforms like SRAM and DRAM, as well as newer non-volatile memory technologies like ReRAM, PCM, STT-MRAM, and SOT-MRAM. Studies showed that even with various levels of quantization and down-scaling of data parameters in CNN algorithms, a satisfactory level of accuracy can still be maintained. This finding opens up opportunities for high-performance and low-power accelerator implementations for IoT and Mobile applications.

2.3.1 DRAM-Based

Li et al. [56] proposed DRISA, a DRAM-based Reconfigurable In-Situ Accelerator architecture that offers powerful computing capability and large memory bandwidth. It uses DRAM memory arrays where every memory bitline can perform bitwise Boolean logic operations. DRISA can compute various functions by combining the functionally complete Boolean logic operations and the proposed hierarchical internal data movement designs. It achieves high performance by simultaneously activating multiple rows and sub-arrays, unblocking the internal data movement bottlenecks, and optimizing activation latency and energy. DRISA achieved an 8.8× speedup and

1.2× better energy efficiency compared with ASICs and a 7.7× speedup and 15× better energy efficiency over GPUs with integer operations.

Later, they improved upon DRISA and proposed SCOPE [57], another DRAM-based in-situ accelerator to accelerate error-tolerant applications like deep learning. To enhance performance and compensate for numerical precision loss, they proposed a stochastic arithmetic optimization method. The proposed architecture provided a 2.3× improvement in performance per unit area compared to the binary arithmetic baseline and a 3.8× improvement over GPU. The proposed H2D arithmetic contributed an 11× performance boost and a 60% numerical precision improvement.

Deng et al. [58] proposed DrAcc, a CNN accelerator based on DRAM technology and utilizes bit operations within the DRAM to implement a ternary weight network, resulting in high inference accuracy. The configuration of data partition and mapping strategies can be adjusted to balance performance, power, energy consumption, and DRAM data reuse factors. DrAcc achieved a frame rate of 84.8 FPS while using 2W of power and delivering 2.9× better power efficiency compared to a process-near-memory design.

Later, they proposed LAcc [59], a PIM accelerator based on DRAM, to support LUT-based multiplication, which is both fast and accurate. By allowing vector multiplication based on LUT in DRAM, LAcc reduced LUT size and increased its reuse. LAcc used a hybrid mapping of weights and inputs to improve hardware utilization. It achieved a rate of 95 FPS at 5.3W for Alexnet, resulting in a 6.3× improvement over the state-of-the-art benchmark.

Ferreira et al. [60] proposed pLUTo, an architecture that utilizes DRAM to enable the parallel storage and querying of lookup tables (LUTs). pLUTo simplifies operations by using low-cost memory reads. pLUTo was evaluated across 11 real-world workloads and outperformed optimized CPU and GPU baselines by an average of 713× and 1.2x, respectively. Additionally, pLUTo reduced energy consumption by an average of 1855× and 39.5x.

2.3.2 SRAM-Based

Eckert et al. [61] proposed Neural Cache, which re-purposes cache structures as compute units for Deep Neural Networks. In-situ arithmetic techniques in SRAM arrays, efficient data mapping, and reduced data movement are proposed. The architecture can execute convolutional, fully connected, and pooling layers in-cache and supports quantization in-cache. Experimental results showed that Neural Cache improves inference latency by 18.3× over a multi-core CPU and 7.7× over a server-class GPU for the Inception v3 model. It also improved inference throughput by 12.4× over CPU and reduced power consumption by 50% over CPU and 53% over GPU.

Later, they improved upon Neural Cache and proposed Eidetic [62], an SRAM-based ASIC neural network accelerator that eliminates the need for frequent off-chip weight loading and minimizes the requirement for intermediate result transfer. The design used in-situ arithmetic in the SRAM arrays and supported various precision types to achieve efficient inference. With the Eidetic architecture, multiple network layers were mapped simultaneously, storing both weights and intermediate results on-chip to avoid the energy and latency costs of off-chip memory access. Eidetic was evaluated on Google's Neural Machine Translation System (GNMT) encoder and demonstrated a 17.20× increase in throughput and 7.77× reduction in average latency compared to a single TPUv2 chip.

Ali et al. [63] proposed IMAC, an in-memory dot product computing primitive using standard 6T SRAM arrays. The accuracy of LeNet-5 and VGG neural network architectures with MNIST and CIFAR-10 datasets was studied for the proposed in-memory dot-product mechanism against circuit non-idealities and process variations. The results showed that the proposed mechanism achieved 99% accuracy for MNIST and 88.8% accuracy for CIFAR-10. The proposed system showed significant improvements compared to the standard von Neumann system, with 6.24× better energy consumption and 9.42× better delay.

Yin et al. [64] proposed XNOR-SRAM, an in-memory computing SRAM macro that performs ternary-XNOR-and-accumulate operations for binary/ternary deep neural networks. XNOR-SRAM simultaneously turns on all 256 rows to accumulate the ternary XNOR operations on the read bitline, which is then digitized with an 11-level flash ADC. The prototype XNOR-SRAM achieved 403 TOPS/W for ternary-XAC operations, with 88.8% accuracy for the CIFAR-10 dataset at a 0.6V supply. XNOR-SRAM provides 33× better energy efficiency and 300× better energy-delay product than conventional digital hardware and delivers a great tradeoff between energy efficiency and inference accuracy.

2.3.3 ReRAM-Based

Chi et al. [65] proposed PRIME to accelerate deep-learning applications in ReRAM-based main memory. The proposed design has an insignificant area overhead and a software/hardware interface for software developers to implement various neural networks. The experimental results showed that PRIME improved performance by approximately 2360× and energy consumption by approximately 895× compared to a state-of-the-art neural processing unit design across the evaluated machine learning benchmarks.

Shafiee et al. [66] proposed ISAAC, an in-memory accelerator for CNN/DNN using ReRAM crossbar arrays that store synaptic weights as analog resistance values. Input bits are sent as analog voltage pulses through the word lines, and the bitline currents represent the accumulated products of input bits with all weights connected along that bitline. These partial products are digitized, shifted, and accumulated to perform complete MAC operations. Weights are represented in 2's complement to account for positive and negative values. ISAAC has a hierarchical architecture including MAC, ADC/DAC, sigmoid, and max pool units in each processing tile. It maps all CNN layers in different tiles and has a pipelined data flow.

Song et al. [67] proposed PipeLayer, a ReRAM-based PIM accelerator for CNN training and testing. The system exploits inter-layer parallelism through a proposed efficient pipeline and intra-layer parallelism through highly parallel design. PipeLayer allows for highly pipelined execution of both training and testing without potential stalls. The system achieved a 42.45× speedup compared to a GPU platform on average and saved an average of 7.17× the energy compared to a GPU implementation.

Tang et al. [68] proposed an RRAM crossbar-based accelerator for inference on Binary Convolutional Neural Networks (BCNN). They discussed the design of BCNN, including the matrix splitting problem and pipeline implementation. Experimental results showed that BCNNs on RRAM had significantly less accuracy loss than multi-bit CNNs for LeNet on MNIST. For AlexNet on ImageNet, the RRAM-based BCNN accelerator saved 58.2% energy consumption and 56.8% area compared to the multi-bit CNN structure.

Qiao et al. [69] proposed AtomLayer, a ReRAM-based accelerator that can efficiently support both CNN training and inference. AtomLayer uses atomic layer computation to eliminate pipeline-related issues and reduce on-chip buffer overhead. To further optimize its performance, AtomLayer uses a unique filter mapping and a data reuse system to minimize layer switching and DRAM access. The experimental results showed that AtomLayer outperforms ISSAC in inference by 1.1× and PipeLayer in training by 1.6x while reducing the footprint by 15x.

Sun et al. [70] proposed an RRAM synaptic architecture, XNOR-RRAM, which uses a bit-cell design to implement XNOR and bit-counting operations. For large-scale matrices, array partition is necessary, and multi-level sense amplifiers (MLSAs) are used to accumulate partial weighted sums. The impact of sensing offsets on accuracy is investigated, and various design options are analyzed. Experimental results with RRAM models and 65nm CMOS PDK showed that the system achieves accuracies of 98.43% for MLP on MNIST and 86.08% for CNN on CIFAR-10. The projected energy efficiency of XNOR-RRAM is 141.18 TOPS/W, showing approximately 33× improvement compared to the conventional RRAM synaptic architecture.

Xue et al. [71] proposed a 2 Mb non-volatile computational-in-memory (nvCIM) macro that combines memory cells and peripheral circuitry based on single-level cell resistive random-access memory devices. This macro has improved capabilities in performing multibit dot-product operations with increased input-output parallelism, reduced cell-array area, improved accuracy, and lower computing latency and energy consumption. Specifically, the macro achieved latencies ranging from 9.2 to 18.3 nanoseconds and energy efficiencies ranging from 146.21 to 36.61 tera-operations per second per watt for binary and multibit input-weight-output configurations.

Wan et al. [72] proposed NeuRRAM, a chip based on RRAM technology that enables the reconfiguration of computational-in-memory cores for different model architectures. NeuRRAM improved energy efficiency compared to previous RRAM-based PIM chips, and its inference accuracy was comparable to software models that use 4-bit weights on various artificial intelligence (AI) tasks, such as image classification and speech recognition. Specifically, NeuRRAM achieved a 99.0% accuracy on the MNIST dataset, an 85.7% accuracy on the CIFAR-10 dataset, and an 84.7% accuracy on Google speech command recognition, as well as a 70% reduction in image-reconstruction error on a Bayesian image-recovery task.

2.3.4 MRAM-Based

Fan et al. [73] proposed a design for a Convolution-in-Memory engine (CIM) using dual-mode SOT-MRAM array architecture. This design enables the implementation of convolution computation within memory to reduce the energy consumption and long data communication distances associated with state-of-the-art CNNs. Later, they proposed an In-Memory Convolution Engine (IMCE) [74] to perform convolution computation within SOT-MRAM memory. IMCE uses parallel computational memory sub-array as a fundamental unit. An accelerator system architecture based on IMCE can greatly reduce energy consumption and accelerate CNN inference. The proposed system architecture can process low bit-width AlexNet on the ImageNet dataset with 785.25 μJ/img, which consumes 3× less energy than the state-of-the-art.

Pan et al. [75] proposed an in-memory computing architecture based on multilevel cell (MLC) spin-toque transfer magnetic random access memory (STT-MRAM) for convolutional operation in Binary Convolutional Neural Networks (BCNNs). The proposed architecture was designed for low current consumption and fully utilized the in-memory computing advantages. The simulation results showed that the proposed architecture reduced power consumption by approximately 35× and 59% compared to the resistive random access memory (RRAM)- and spin-orbit torque-STT-MRAM-based counterparts, respectively, on the MNIST dataset.

Patil et al. [76] proposed an MRAM-based architecture called MRAM-DIMA to perform multi-bit matrix-vector multiplication for deep neural networks. The MRAM-DIMA used a standard MRAM bit-cell array and achieves lower energy and delay compared to conventional digital MRAM architecture with a 4.5× and 70× reduction, respectively. The impact of circuit non-idealities, including process variations, on DNN accuracy was estimated using behavioral models. For LeNet-300-100 on the MNIST dataset and a 9-layer CNN on the CIFAR-10 dataset, the MRAM-DIMA can tolerate a 24% and 12% variation in cell conductance and still achieve accuracy drops of $\leq 0.5\%$ and $\leq 1\%$, respectively.

Angizi et al. [77] proposed CMP-PIM, a SOT-MRAM-based accelerator to execute a hardware-oriented comparator-based neural network named CMPNET. CMPNET replaces the multiplications in convolution layers with more efficient and less complex comparison and addition. Later, they proposed MRIMA [78] proposed MRIMA, an STT-MRAM-based in-memory accelerator for efficient in-memory computing. MRIMA transforms STT-MRAM arrays into massively parallel computational units capable of working as nonvolatile memory and in-memory logic. It exploits bitline computing methods to implement complete boolean logic functions between operands within a memory array in a single clock cycle. The device-to-architecture co-simulation results showed that MRIMA can obtain 1.7× better energy efficiency and 11.2× speed-up compared to ASICs for CNN acceleration. MRIMA also shows 77% and 21% lower energy consumption compared to CMOS-ASIC and domain-wall-based designs, respectively, for AES in-memory encryption.

2.3.5 FPGA Block-RAM-Based

The trend towards PIM architectures is inspiring new reconfigurable fabrics that integrate bit-serial arithmetic units into FPGA Block-RAM (BRAM) IP to form PIM tiles. This trend is relatively recent in the FPGA community. Thus, only a few works have been published to date.

Inspired by Neural Cache [61], Wang et al. [79] proposed Compute-Capable BRAMS (CCB) adding in-memory compute capabilities to FPGA BRAMs, allowing them to act as storage units or execute bit-serial arithmetic operations. This results in a 1.6× and 2.3× increase in peak multiply-accumulate throughput for a large Stratix 10 FPGA, at a minimal cost of only a 1.8% increase in the FPGA die size. The proposed change does not affect the BRAM's interface to the programmable routing. A reconfigurable in-memory accelerator architecture (RIMA) was developed for deep-learning inference, which uses CCBs and FPGA's reconfigurability to achieve 1.25× and 3× higher performance for 8-bit integer and block floating-point precisions, respectively, compared to the state-of-the-art Brainwave deep-learning soft processor. RIMA implemented on a Stratix 10 FPGA with compute-capable BRAMs achieved an order of magnitude higher performance than a same-generation GPU.

Improving upon CCB, Arora et al. [80] proposed modification to FPGA BRAMs to create CoMeFa (Compute-In-Memory Blocks for FPGAs) RAMs, which provides compute-in-memory capabilities by combining computation and storage in one block. CoMeFa RAMs use the true dual-port nature of FPGA BRAMs and include multiple programmable single-bit bit-serial processing elements. They can be used for computing with any precision, making them valuable for applications like deep-learning. They proposed two variations of CoMeFa RAMs: CoMeFa-D, which is optimized for the delay, and CoMeFa-A, which is optimized for the area. Compared to existing proposals, CoMeFa RAMs are practical to implement and versatile, finding applications in various parallel applications like Deep Learning, signal processing, and databases. By adding CoMeFa-D or CoMeFa-A RAMs to an Intel Arria-10-like FPGA, a geomean speedup of 2.55× or 1.85x was achieved, respectively, with algorithmic improvements and efficient mapping, at the cost of 3.8% or 1.2% area, respectively.

Both CCB and CoMeFa used a transposed data layout for bit-serial computation, where operands are stored along the bitlines occupying multiple wordlines in a column-major format. This adds additional latency to the end application, due to conversion to and from the row-major format. To solve this problem, Chen et al. proposed a hybrid bit-serial and bit-parallel Multiply-Accumulate (MAC) dataflow in BRAMAC [81] and M4BRAM [82]. In contrast to CCB and CoMeFa, this approach avoids computing on the primary SRAM array, which is large and thus slow and power-intensive. Instead, they copy the operands from the BRAM to a smaller, separate “dummy array” for MAC operations. This approach liberates the BRAM data ports for read/write tasks, facilitating the concurrent execution of MAC operations, data loading, and parallel DSP operations. BRAMAC’s constraint lies in necessitating uniform precision (2-/4-/8-bit) for both weights and activations, restricting its use to uniform-precision DNNs exclusively. M4BRAM overcomes this limitation by enabling variable activation precision, spanning from 2 to 8 bits, while maintaining a MAC latency that scales linearly.

2.4 Benchmark Design

This research work has its roots in the Ph.D. dissertation of Dr. Atiyehsadat Panahi [3]. In her dissertation, she developed a prototype of a memory-centric FPGA overlay, which is designed using a SIMD processor array to support rapid coding, programmability, and flexibility for the acceleration of machine learning applications. The developed architecture was 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 memory-centric architecture, a set 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 the SIMD architecture. The studies presented in [3, 25, 29, 31] show significant speed-up compared to existing custom designs ($1.75\times$) and HLS ($24.51\times$). They offer massive parallelism in a mid-sized Virtex-7 FPGA (xc7vx485) by packing 10K-16K PEs. The overlay architecture presented in [3, 25, 29] has been adopted as a benchmark design for comparative study.

Though the basic architecture of this research work is similar to the benchmark design, the design approach and the end goals are vastly different. Table 1 compares the benchmark design and this work to distinguish between them. The first four criteria in the table are the same in both works; they both are FPGA overlays based on SIMD array processors. Both employ PIM blocks to carry out computation and can be programmed using software tools. However, the similarity ends there. Though the benchmark is an overlay that can be programmed to implement any

Table 1: Comparison of this research work with the benchmark design

Criterion	Benchmark Design	This Work
FPGA Overlay	Yes	Yes
SIMD Array Processor	Yes	Yes
PIM Overlay	Yes	Yes
Software programmable	Yes	Yes
Reconfigurability	Dimensions only	Reconfigurable Architecture
Data network	NEWS, Bin-Add (Fixed)	Custom, Binary-Hopping
Control Network	Broadcast (Fixed)	Custom, Pipelined Tree
Front-end interface	Memory-Mapped Registers	Custom, Abstract ISA
Component Architectures	Fixed	Plug-and-Play
Extensible	No	Yes
Customization Access	Software (SW) level only	Both SW and HW levels
Design Goals	Low-Latency (qualitative)	Ideal Design Standard
Performance Goal	Unspecified	BRAM max frequency
Scaling with BRAM	No	Linear Scaling
Reduction Latency Goal	Unspecified	Balanced

deep-learning model, it does not aim for the high-performance and scalability memory-centric design goals achieved in this dissertation.

The second section of Table 1 summarizes the limitations of the architecture of the benchmark design. The benchmark design has fixed networks for data movement and control signals. The front-end interface is a set of memory-mapped registers, which directly connects to the PE blocks without any intermediate networking mechanism. This drastically deteriorates the system performance achieving only 25% of the maximum achievable system frequency [29]. On the other hand, this work provides a flexible system-level architecture with well-defined interface to support custom implementations of each of these components of the accelerator to achieve maximum performance on a target platform.

The third section of the table highlights the customization capabilities offered by this work, not offered by the benchmark design. The only part of the benchmark design that can be reconfigured for a target application or device is the dimensions of the array; the designer can specify how many rows and columns of PEs to implement and how to group them in the tiles. The accelerator developed in this work offers a fully reconfigurable architecture, where any part of the accelerator can be reconfigured or replaced as per the designer's choice seamlessly. Each component in the benchmark design is fixed and cannot be changed without breaking other parts of the system. Thus, this prevents any attempt to perform application-specific optimization on the benchmark design. The only way to customize the benchmark overlay is through software programming using its custom instruction set. In the architecture developed in this research work, the functionality of each of the components is well-defined without restricting it to a particular implementation. This enables a plug-and-play approach to prototyping and rapid accelerator development. Consequently, it can make application-specific optimizations as easy as writing a configuration file.

The last section of the table highlights the fact that, the benchmark design only aimed to reduce latency without establishing any quantifiable design goal. This work targeted and met very specific memory-centric design goals for attaining the highest performance and scalability. These design goals define the theoretical upper limits of the PIM array-based FPGA accelerators. On the other hand, the benchmark design does not define or meet these goals in any of its configurations.

Chapter 3

Defining The Ideal Design Standard

Processing-in-Memory (PIM) architectures offer promising features that have the potential to significantly enhance computing performance by integrating processing capabilities directly into memory. Despite these advantages, the proposed PIM architectures to date, including the ones discussed in the last chapter, strive to achieve optimal performance but face inherent limitations. While compute density is increased, the maximum BRAM clock frequency is significantly reduced. Additional scalability issues are also introduced at the system level: a faster and larger device does not imply a faster system speed or a linear increase in the compute units with increased BRAM density. While each proposal offers some relative improvement over prior designs, there is currently no absolute metric or yardstick that could be used to evaluate the efficiency of these designs. This has made it difficult to perform quantitative comparisons between PIM arrays implemented in different logic families from one vendor or between different vendors.

These limitations often stem from a lack of a unified design standard, resulting in inconsistent performance and scalability challenges across different implementations. No existing PIM architecture fully addresses these crucial aspects, leaving a gap in the field. In this chapter, we aim to bridge this gap by defining a comprehensive design standard for PIM architectures. This standard is intended to serve as an absolute metric for comparing the efficiency of existing designs while also serving as an aspirational set of design objectives for future designs. The feasibility of each goal and related implementation challenges are also discussed. Through this, we hope to provide a robust framework that future PIM designs can adhere to, facilitating the development of more efficient and effective architectures.

3.1 Analysis of Existing PIM Designs

Table 2 summarizes the maximum achieved frequencies of existing PIM designs. From the relative frequency columns, it is observable that the clock frequency f_{PIM} of all the PIM designs,

Table 2: Maximum Frequency (MHz) of Existing FPGA-PIM Designs

PIM Design	Type	Device	f_{BRAM}	f_{PIM}	Relative	f_{Sys}	Relative
CCB	Custom	Stratix 10	1000	624	62%	455	46%
CoMeFa-A	Custom	Arria 10	730	294	40%	288	39%
CoMeFa-D	Custom	Arria 10	730	588	81%	292	40%
BRAMAC-2SA	Custom	Arria 10	730	586	80%	-	-
BRAMAC-1DA	Custom	Arria 10	730	500	68%	-	-
M4BRAM	Custom	Arria 10	730	553	76%	-	-
SPAR-2	Overlay	UltraScale+	737	445	60%	200	27%
PiMulator	Overlay	UltraScale+	737	-	-	333	45%

both overlays and custom BRAMs, are significantly slower compared to the maximum frequency for the device BRAMs (f_{BRAM}). Their system frequencies (f_{Sys}) are $2.1\times - 3.7\times$ slower than the BRAM maximum frequencies (f_{BRAM}). This decrease in system frequency was attributed to the limitations of the soft logic and the routing resources of the FPGAs. It was also reported as unlikely that an FPGA accelerator at the system level would operate at a frequency surpassing the degraded frequency (f_{PIM}) of these PIM designs, even in a more advanced node than the evaluation platforms [1, 81–83].

Further observation yielded that most of these systems could not utilize all available BRAMs as PIMs. This lower utilization combined with a lower clock frequency results in less efficient use of the available internal BRAM bandwidth of the devices and a lower system-level compute density. A final observation shows a troubling common pattern: as the utilization of BRAMs increases the achievable system-level clock frequency decreases [1, 84].

These observations motivated our interest in understanding if these results were a new reality of BRAM PIM arrays or symptomatic of specific design decisions and implementation choices. This led us to ask two questions. What is the fastest frequency a PIM-based design should/could be able to achieve on FPGAs, and was it possible to scale compute density up to the maximum BRAM capacity without degrading the clock frequency?

We posited that the compute efficiency of PIM-Arrays should be measured relative to an FPGA's full internal bandwidth and only limited by the density of the devices BRAMs. This would require a PIM architecture to run at the BRAM maximum speed with compute density

Table 3: Delay (ns) Breakdown of 1-level Logic Path in AMD Devices

	FF-C2Q[1]	LUT[2]	FF-Setup	Total[3]	BRAM[4]	Net Budget	Min[5]
V7	0.290	0.34	0.255	0.885	1.839	0.954	0.272
US+	0.087	0.15	0.098	0.335	1.356	1.021	0.102

[1] Clock-to-Q delay of flip-flops
[2] Average of delay through LUTs
[3] Total cell delay
[4] BRAM pulse-width requirement, clock period for Fmax
[5] Minimum net delay through a switchbox

scaling linearly up to the full density of BRAMs on a device. We termed such a PIM architecture as the Ideal Design Standard.

3.2 Ideal Clocking

In FPGAs, BRAMs are the single component with the longest latency [85–87], thus representing the timing bottleneck for setting clock speed. Though in a typical FPGA design, the logic and routing delays can dominate the overall path delays, at the unit level, FPGA resources like LUTs, FF, and routing blocks are much faster than the BRAM. Thus, we define the maximum frequency (Fmax) of the BRAM as the Ideal Standard clock frequency for the PIM accelerator. This requires processing elements along bit lines to be designed such that they do not degrade this frequency.

To assess the practicality of this standard, we closely examined two AMD FPGA families: Virtex-7 and UltraScale+. While Virtex-7 CLB resource's delay numbers are available in the datasheet [85], those numbers are not publicly available for UltraScale+ devices. Thus, we created a design where all timing paths are one logic level deep and averaged all paths to obtain Table 3. The Total column sums the cell delays in the columns to its left. The BRAM column lists the clock period for BRAM Fmax. The Min column displays the minimum delay of a net passing through a switchbox. Net Budget is derived by subtracting the Total column from the BRAM column. Comparing the net budget with the minimum net delay shows the possibility of designing at least two LUTs deep logic paths that can run at the BRAM Fmax on these device families.

In certain FPGA families, achieving this constraint may be challenging due to the presence of multiple dies and fixed-function blocks supporting various functionalities, such as PLLs, high-performance IO blocks, PCIe blocks [88], and application processors [89]. These blocks impact placement and routing, eventually affecting the maximum achievable clock frequency. In such devices, this Ideal Standard may not be achievable. However, FPGA vendors can employ this clocking standard to ensure the achievability of this constraint in device families targeting PIM designs.

3.3 Ideal Clocking Design Challenges

There are several challenges in achieving this Ideal Standard for clock speed, and most of them can be addressed at the architectural level. The most important design consideration to make is the logic depth design. The part of the system logic residing in the same clock domain as the BRAM should be at least as fast as the BRAM. This may require adding optional pipelines in the design, which can be enabled at a later stage of implementation if the logic depth is limiting the clock speed.

Another major problem is the high fanout nets. Most of the PIM array-based designs share the control logic across multiple PIM blocks [3, 29, 84]. These nets carrying the control signals can have several hundreds to thousands of fanouts, which need to be managed carefully to achieve the Ideal Standard. This fanout can be kept low by replicating the control logic. If large fanout is unavoidable, then pipelined fanout trees need to be synthesized for these signals.

Despite the aforementioned design considerations, timing failures may still occur during placement and routing due to long routes causing excessive net delays. To mitigate this, the system architecture should employ a tile-based approach at the system level [1, 29, 83, 84] to localize logic and routing. The RTL design should be implemented with an awareness of potential placement and routing issues. If feasible, alternative implementations should be incorporated to be selected during the implementation stage to address placement and routing issues.

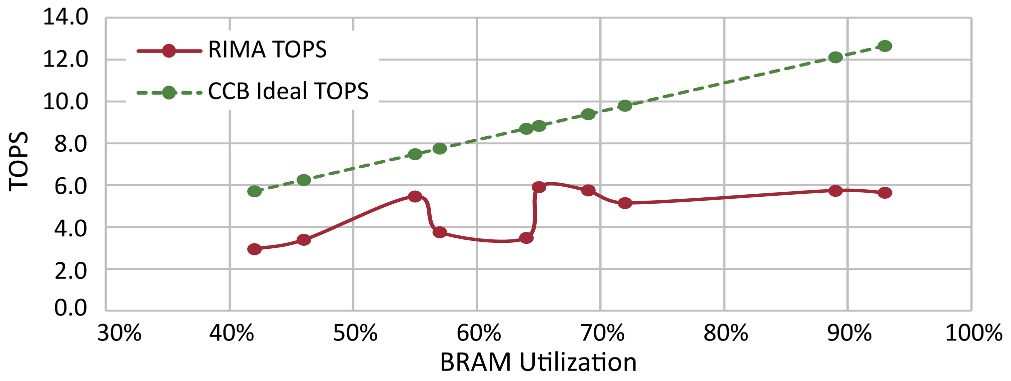


Figure 1: Ideal scaling vs. actual TOPS of RIMA on Stratix 10 GX2800

3.4 Ideal Scaling of Peak-Performance

One of the major benefits of the PIM architecture is the massive parallelism it can offer: all bitlines of the memory array can be designed into concurrent processing elements [1,83,84,90]. It is desired that the compute capacity should scale proportionately with the number of available BRAMs. As an Ideal Standard, we posited that the peak-performance of a PIM design needs to scale linearly with the on-chip BRAM count. Existing PIM designs do not adhere to this standard.

The compute capacity in custom-BRAM-based PIM designs [1,81–84] scales linearly with BRAM count if all BRAM tiles are used in PIM mode. However, a significant sacrifice is imposed in the clock frequency that ends up limiting the achievable peak-performance on the device. The f_{PIM} column in Table 2 indicates that the custom-BRAM PIM designs run up to 2.5× slower than the BRAM Fmax. The situation is worse at the system level. As reported in Table-II of [84], CCB-based accelerator RIMA experiences decreased system speed as the BRAM usage increases. A similar trend is observed in CoMeFa-based designs as reported in Tables 8 and 9 of [1]. Fig. 1 plots the peak-performance of RIMA using Table-II of [84]. The peak-performance of the system is computed from the reported BRAM utilization of Stratix 10 GX 2800 FPGA and the corresponding frequency of the M-DPE clock domain. We can see that the peak-performance of RIMA shows an irregular trend. This irregularity can be attributed to the system-level architecture of RIMA. If RIMA was designed following the ideal scaling standard, even at the degraded CCB frequency of 624 MHz, its peak-performance would align with the CCB Ideal

TOPS line. The gap between these plots represents wasted compute capacity and memory bandwidth provided by CCB BRAMs.

3.5 Ideal Scaling Design Challenges

The ideal scalability in custom-BRAM-based PIM systems can be achieved in most cases by keeping the system clock at BRAM Fmax avoiding too deep combinatorial logic and routing issues as discussed before. Apart from the clock speed, the main challenge preventing these PIM-based accelerators from achieving the ideal scaling is the high logic utilization of the rest of the system. The same design principle as in the ideal clocking standard must be applied here as well: BRAM should be the utilization bottleneck, not the control logic.

In PIM accelerators, PIM blocks handle computation, while the rest of the system primarily manages data pipelines and control logic [1, 29, 83, 84]. To prevent the control logic from becoming the utilization bottleneck, controllers should be shared between multiple PIM blocks. Control signals should be designed to minimize the number of unique control sets in the system. A control set is a group of control signals (set/reset, clock enable, and clock) for a register or latch [91]. Excessive unique control sets can degrade placement quality, impacting system scalability and clock frequency. Overall, the rest of the system should complement the PIM array without limiting its scalability and performance.

3.6 Ideal Reduction Latency

Reduction is a critical operation in applications like GEMM and deep-learning as it requires data movement throughout the distributed BRAM memories and/or processing elements of the PIM array. Accumulation, a common reduction operation, is often implemented using a pipelined adder tree [1, 84] as shown in Fig. 2. Adder trees are resource-hungry, especially on routing, requiring more adders and routing resources as the PIM row size increases. Bit-parallel implementations are even harder to manage compared to bit-serial: INT8 requires 8× more logic and routing resources than bit-serial implementations. Such high utilization can subsequently

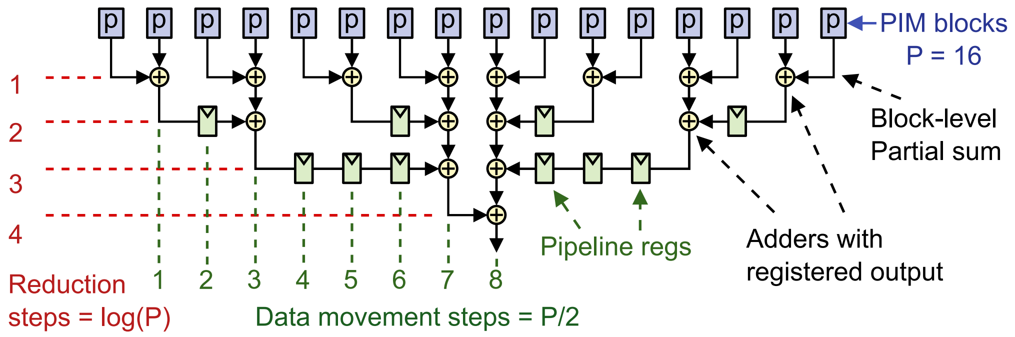


Figure 2: Reduction latency breakdown of a pipelined binary adder tree maintaining ideal clocking constraint.

affect system frequency and scalability.

To prevent the reduction network from becoming the utilization and routing bottleneck, sharing logic and routing resources with the rest of the system may be necessary. Existing high-performance reduction architectures for FPGA implementation [92–97] highlight the unavoidable trade-off between reduction latency and resource utilization. To guide this trade-off, we propose the Ideal Standard for reduction latency as the following empirical model,

$$\text{Array-Level Reduction}_{\text{ideal}} = aN\log P + bP + c \tag{1}$$

$$\text{In-Block Reduction}_{\text{ideal}} = aN\log k \tag{2}$$

In this context, a PIM "block" is a single BRAM tile, like CCB, CoMeFa, BRAMAC, etc., along with its related logic. In-Block reduction generates partial sums accumulating the PEs in a PIM block, while array-level reduction accumulates these partial sums. Here, P = no. of partial sums obtained from all PIM columns involved in the reduction process, N = operand width (precision), k = no. of PE columns in a PIM block; a, b, c are implementation-specific parameters with their ideal range specified in Table 4.

The intuition behind (1) and the parameter ranges can be explained using Fig. 2. The total reduction (accumulation) latency can be broken down into two parts: reduction operation (add) and data movement. The term $aN\log P$ represents the latency of reduction operations (add) only, requiring at least $\log P$ reduction steps; the base of the log represents the number of operands

Table 4: Parameters of Ideal Standard for Reduction Latency

Parameter	Ideal Range	Related to
a	$1/\mathrm{N} \leq a \leq 2$	Latency of reduction steps (addition)
b	$0 \leq b \leq 1$	Latency of data movement
c	$0 \leq \mathrm{c}$	Cycles spent outside reduction network

reduced per step, typically 2. The PE architecture (bit-serial, bit-sliced, or bit-parallel) determines the value of a. The lower bound of a is 1/N because at least one cycle is needed per reduction step ($aN \geq 1$). We set the upper bound for a to 2 because bit-serial PEs, especially in overlays, commonly require 2 cycles to process each bit ($aN = 2N$) of the operand [2, 3, 29].

Under the ideal clocking constraint, data-movement latency depends linearly on the number of PIM columns in a row, represented by bP. Assume each PIM block (p) in Fig. 2 occupies a large enough area such that a net does extend beyond two consecutive PIM blocks without violating the ideal clocking constraint. Then constructing the adder tree without sacrificing the clock speed requires one pipeline stage to be placed along each PIM column as in Fig. 2. This requires P/2 cycles ($b = 1/2$) for accumulation towards the middle in this example. In general, the value of b depends on the speed of FPGA's routing resource relative to BRAM Fmax. Typically $b < 1$ in modern FPGAs with fast routing resources, which allows nets to span multiple PIM columns at BRAM Fmax. The lower bound of b is 0, corresponding to a bit-parallel pipelined reduction tree that perfectly overlaps data movement with computation.

In (1), c represents delays outside the reduction network. The In-Block partial-sum generation latency (2) is a simpler version of (1), where k denotes the number of PE columns accumulated within the PIM block. A term for pipelined data movement is absent as signals can move within a block without violating the ideal timing constraint. In-Block reduction latency (2) can be absorbed into c, making (1) as the overall reduction latency standard. The lower bound of c is 0, representing no additional cycles outside of the reduction network. Having no upper bound allows the network architecture to vary without violating the proposed Ideal Standard; some preprocessing steps may be employed such as the pop-count-based adder in RIMA [84], before

Table 5: Reduction/Accumulation Latency of Existing PIM Designs

	Block Level	PIM Array Level	Utilization[1]
Linear-Add[2][3]	3N (k-1)	3N (P-1)	Low
Binary-Add[2][3]	2Nlog(k) + N(k-1)	2Nlog(P) + N(P-1)	Low
CCB/CoMeFa [84]	2Nlog(k) + $\log^2$(k)	log(P) + 2	High
Proposed Standard	*a*Nlog(k)	*a*Nlog(P) + *b*P + *c*	Balanced[3]

[1] Qualitative logic and routing resource utilization
[2] NEWS network with linear shift then add
[3] Offers latency vs. resource utilization trade-off

entering the reduction network that requires tens or hundreds of cycles.

Though the proposed standard (1) closely resembles a pipelined adder tree, it can provide insights and help identify design inefficiencies in various other architectures. Importantly it can steer designers towards optimal or near-optimal reduction network designs. We explore this in Chapter 5 through a quantitative study of existing designs.

3.7 Ideal Reduction Design Approaches

Table 5 shows the latency equations for bit-serial PIM architectures [1, 2, 29, 84] and their relative utilization of resources. The bit-parallel architectures [81, 82] are not listed because the block-level reduction is built into their MAC units and they do not have a proposed implementation for array-level reduction.

In SPAR-2 [3], the reduction is performed using a NEWS network with two different approaches. In each iteration of the linear-shift linear-add approach, partial sums are bit-serially shifted to the next column, added with the multiplication result of that column, and then shifted again. In each iteration of the linear-shift binary-add approach, partial sums are bit-serially shifted through a given number of intermediate columns, then added with the partial sum stored in that column, and then shifted again doubling the number of intermediate columns. Thus, the binary-add approach has the log term, unlike the linear-add approach. The block-level and array-level latency equations are exactly the same because, the operands move through PE columns connected using the NEWS network, ignoring the block-level grouping of the PEs. The NEWS network is the simplest of the data movement networks among the designs listed in the table, requiring very small logic and routing resources.

Accumulation in CCB and CoMeFa [1, 84] at the block level is implemented as a series of across-bitline copies and additions, followed by a pop-count-based adder. Unlike other designs in Table 5, it has a $\log^2$ term at the block level because it increases the number of bits accumulated after each addition, to increase the precision of the accumulation result. They mentioned using a "global reduction tree" at the array-level, without discussing its implementation details. If we assume the fastest implementation, the global reduction tree can be implemented using a pipelined adder tree with the array-level latency represented by the CCB/CoMeFa row of Table 5 with a pipeline overhead of 2 cycles. This implementation consumes the highest logic and routing resources, which can significantly affect the scalability of a PIM array with hundreds of blocks per row in the array.

As observed from the table, each of the architectural improvements brings the latency expression closer to the proposed standard (1). This standard latency equation can be used as a guide to design the optimal trade-off between latency and utilization. It can also be used to evaluate and compare different reduction network architectures by approximating their latencies using the model. If the latency expression of an implementation deviates too much from the Ideal Standard, like the NEWS network-based implementations, this will mean that the network architecture is not optimal for reduction and adjustments are needed. In Chapter 5, we will quantify these latencies and demonstrate how the proposed standard can help us identify the design inefficiencies and guide us toward an optimal design.

Chapter 4

PiCaSO: A Processor-in-Memory Scalable and Fast Overlay

After establishing the proposed Ideal Design Standard, we created a PIM block that adheres to the standard. PiCaSO, a Processor in/near Memory Scalable and Fast Overlay, is a representative PIM overlay architecture that served as a practical test of the validity of the standard, revealing both the potential and the challenges inherent in meeting these ambitious goals. Successfully meeting these goals resulted in a high-performance PIM block, demonstrating not only the feasibility of the standards but also their effectiveness in guiding the development of superior PIM architectures. PiCaSO has been published at [98] as an open-source implementation and is freely available for study, use, modification, and distribution without restriction.

4.1 PIM Architecture

A generic bit-serial PIM block has a datapath that starts at the data output ports of the BRAM, passes through the logic elements that usually implement the ALU, and then returns to the data input ports of the BRAM [1, 29, 99]. However, as the operands are streamed on separate bit-lines this datapath cannot support reduction of operands in different PEs. If the PIM block does not have a built-in mechanism for such reduction, the operands need to come out of the memory block, be processed in the LBs of the FPGA, and then written back to the target PE column of the memory block. This defeats the purpose of having a PIM block.

To address this shortcoming, we introduce an operand-multiplexer (Op-Mux) module between the BRAM and the ALU. The Op-Mux implements the mechanism of multiplexing the bit lines going into the bit-serial ALUs. To overlap data movement with computation, we design a network module that can move the bit-serial operand stream over the network and directly feed it to the ALU for reduction operations. Fig. 3 shows the proposed architecture of PiCaSO.

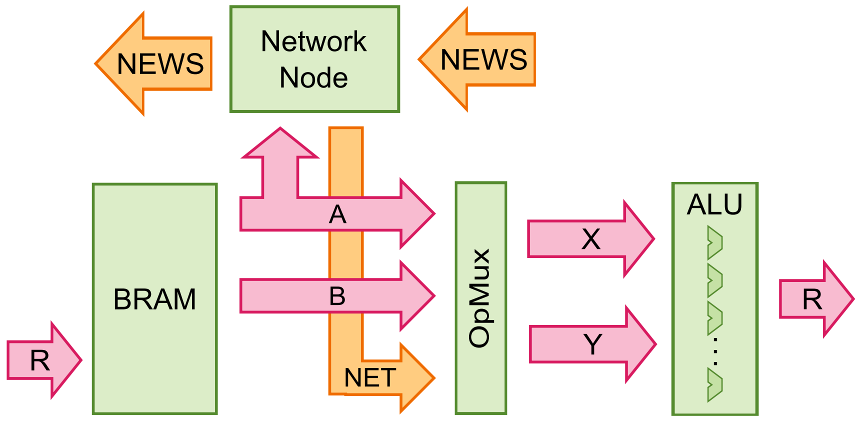


Figure 3: Proposed PIM architecture

4.1.1 Register File

In our proposed design, each column of a BRAM acts as a bit-serial register file. One important, but less discussed, aspect of the register file design using BRAMs is to determine the optimal dimension of the PE-Block. A square array may seem reasonable: a PE-Block with 16 PEs can be logically organized into a 4×4 array [29]. However, it may not be optimal depending on the target application and the FPGA. To get the best performance out of a design implemented on an FPGA, its logical structure needs to map well to the physical layout of the FPGA. In Virtex-7 and Virtex UltraScale+ FPGAs, there can be 10–16 BRAM columns, each containing 60–120 of BRAMs. For such a layout, an array processor with a wider PE-Block will map better than a square PE-Block. For this reason, we designed a PE-Block with a logical array dimension of 1×16 targeting a Virtex FPGAs. This also reduced the number of PE-Blocks per row, thus reducing the number of hops while moving data through the network for accumulation.

4.1.2 Bit-Serial ALU

In array-processor applications, the most frequent operation is multiply-accumulate followed by the addition of the bias vector. The accumulation part requires data movement, which is handled by the data network and the Op-Mux module. ALU is responsible for performing the multiplication and addition. In a bit-serial PE, multiplication is implemented through repeated

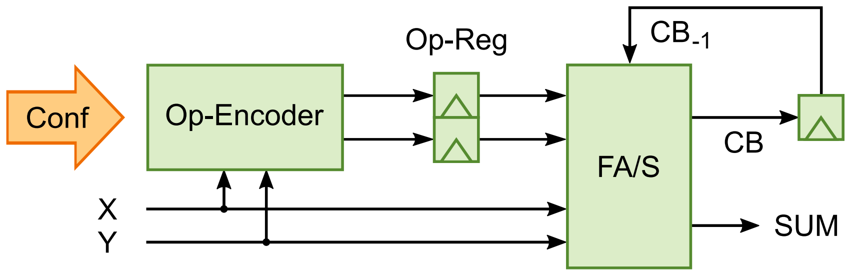


Figure 4: Architecture of the bit-serial ALU of PiCaSO

Table 6: ALU (FA/S) Op-Codes

Op-Code	Output (SUM)	Description
ADD	X + Y	Acts as a Full-Adder (FA)
SUB	X - Y	Acts as an FA with borrow logic
CPX	X	Copies operand X unmodified
CPY	Y	Copies operand Y unmodified

ADD/SUB operations following algorithms like Booth's Radix-2 or Radix-4.

Another very common operation performed in CNNs is min/max pooling, which involves the comparison of two or more operands and the selection of the min/max of them. Bit-serial comparison can be performed by comparing two operands serially from MSB to LSB. The first mismatched bit determines the comparison result (X or Y) and the rest of the bits of the selected operand are simply copied from that bit onwards. So, the ALU needs to have a mechanism for selecting one of the operands unmodified.

To support these essential operations and to optimally map the logic into FPGA, we selected four fundamental operations shown in Table 6 for the ALU. ADD and SUB can be used as standard arithmetic operations as well as a step of the multiplication operation to update the partial product. CPX and CPY can be used to implement min/max pooling or any other filter-like operations that requires selecting one of the two operands. CPX is also used in multiplication as a substitute for the no-operation (NOP) step of Booth's algorithm.

Fig. 4 shows the architecture of the bit-serial ALU. It consists of two logical parts: Full-ADD/SUB module (FA/S) and op-code encoder (Op-Encoder). FA/S implements the four operations in Table 6. It is a pure combinatorial block with two single-bit outputs: SUM and CB,

Table 7: Op-Encoder Configurations for Booth's Radix-2 Multiplier

Conf	YX	ALU Op-Code	Description
000	xx	ADD	Request ADD
001	xx	CPX	Select X operand
010	xx	CPY	Select Y operand
011	xx	SUB	Request SUB
1xx	00	CPX	NOP
1xx	01	ADD	+Y
1xx	10	SUB	-Y
1xx	11	CPX	NOP

implemented as the following logic functions,

$$\mathrm{SUM} = \mathrm{X} \oplus \mathrm{Y} \oplus \mathrm{CB}_{-1}, \quad \text{if ADD or SUB}$$

$$\mathrm{CB} = \begin{cases} \mathrm{X} \cdot \mathrm{Y} + \mathrm{X} \cdot \mathrm{CB}_{-1} + \mathrm{Y} \cdot \mathrm{CB}_{-1}, & \text{if ADD} \\ \overline{\mathrm{X}} \cdot \mathrm{Y} + \overline{\mathrm{X}} \cdot \mathrm{CB}_{-1} + \mathrm{Y} \cdot \mathrm{CB}_{-1}, & \text{if SUB} \end{cases}$$

Op-Encoder provides an abstract interface to the FA/S module for high-level operations. It takes a configuration code (Conf) from the controller, determines the corresponding Op-Code for FA/S for that configuration, and stores it in a register (Op-Reg). In PiCaSO, we use it as an interface to implement Booth's Radix-2 multiplication. Table 7 shows the mapping of the 3-bit configuration code to the Op-Code of the FA/S module. If the MSB of the Conf is 0, a requested Op-Code is loaded into Op-Reg. If the MSB is 1, the Op-Code loaded depends on the YX 2-bit combination. If YX holds the 2-bits of the multiplicand for an iteration of Booth's Radix-2 algorithm, the selected Op-Code maps to Booth's encoding to update the partial product. The Op-Encoder can be modified to add other high-level operations, such as pooling, without redesigning the FA/S module.

The FA/S module has 5 input bits: X, Y, CB_{-1}, and 2-bit Op-Code, and 2 output bits: SUM and CB. This can be mapped to a single LUT6 (2×LUT5) of a Virtex-7 FPGA. The Op-Encoder module also has 5 input bits and 2 output bits. Therefore, this can also be mapped to a single

LUT6. In addition to that, 2 flip-flops are needed to store the Op-Code generated by the Op-Encoder, and 1 flip-flop is needed to store the CB output of FA/S to be used as CB_{-1} in the next cycle. Thus, the entire ALU can be implemented in less than a logic slice, using 2 LUTs and 3 flip-flops. From the data stream point of view, the Op-Encoder is not part of the datapath. Thus, the ALU acts as a single LUT stage in the datapath.

4.1.3 Operand Multiplexer

The Op-Mux module is used to perform reduction operations between the PEs in a PE-Block, as well as to overlap computation and data movement during reduction. To achieve this goal, Op-Mux module provides the interface to route the network stream directly into the ALU. As shown in Fig. 3, the Op-Mux module takes in the bit-streams coming out of the network module and BRAM ports, then streams the multiplexed output as X & Y operands of the ALU.

In traditional NEWS networks, reduction works by moving data from a PE register to a register of its neighboring PE, then performing the required operation on that register and another register in the destination PE [25,29]. In this approach, q PEs in a row will take $q-1$ moves. For N-bit operand, a move takes N-cycles if we use 2 cycles for read–write and take advantage of the dual-port configuration of BRAMs. Each ADD/SUB takes 2 cycles to process each bit and so, takes $2N$ cycles to process each ADD/SUB operation. For q PEs, it needs at least $\log_2 q$ such operation. Thus, we can represent the PE-Block reduction latency (PB-RL) using the following equation,

$$\text{PB-RL}_{\text{NEWS}} = N \cdot (q - 1 + 2\log_2 q) \tag{3}$$

Op-Mux module significantly reduces this latency using a folding technique. Fig. 5 shows two types of folding patterns for a PE-row with 8 columns, which can be implemented by the Op-Mux module. In pattern (a), after adding an operand with its fold-1 pattern, PE 0, 1, 2, and 3 contain the summation of 0 & 4, 1 & 5, 2 & 6, and 3 & 7 respectively. In pattern (b), after adding an operand with its fold-1 pattern, PE 0, 2, 4, and 6 contain the summation of 0 & 1, 2 & 3, 4 & 5, and 6 & 7. In both cases, after applying fold-1, fold-2, and fold-3 in that order, the accumulation

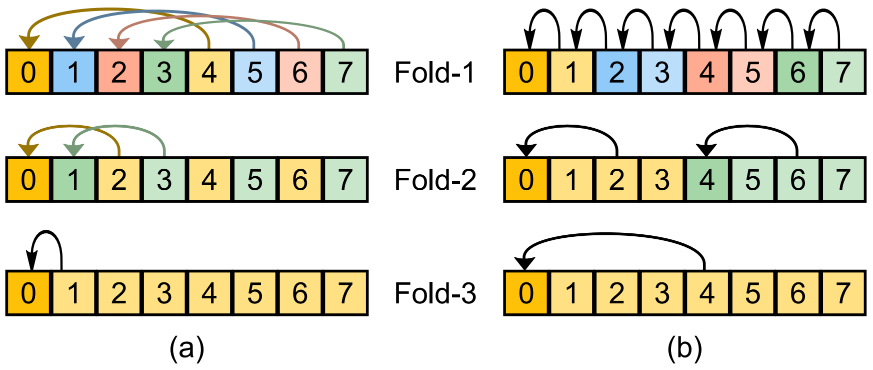


Figure 5: Folding patterns in Operand Multiplexer

Table 8: Configurations of Operand Multiplexer

Config Code	X	Y	Description
A-OP-B	A	B	Used in standard operations
A-FOLD-1	A	{0, A[H2]}	A[H2]: second half of A
A-FOLD-2	A	{0, A[Q2]}	A[Q2]: second quarter of A
A-FOLD-3	A	{0, A[HQ2]}	A[HQ2]: second half-quarter of A
A-FOLD-4	A	{0, A[HHQ2]}	A[HHQ2]: second half of A[HQ1][1]
A-OP-NET	A	NET	Operates on network stream
0-OP-B	0	B	Used in the first iteration of MULT

[1] A[HQ1] : first half-quarter of A

result will be stored in PE-0. Fold-1 of pattern (b) can be especially useful in CNN models, where each PE needs access to its adjacent PEs. Such a folding scheme can be implemented using the multiplexers at the output of SA in custom PIM blocks as in [1].

To reduce q PEs in a row will take $\log_2 q$ folds. Thus, we can represent the PE-Block reduction latency using Op-Mux using the following equation,

$$\text{PB-RL}_{\text{opmux-rw}} = 2N \cdot \log_2 q \tag{4}$$

This eliminates the linear term from (3) related to the data movement through the NEWS network.

In the proposed design of PiCaSO, we select the seven configurations shown in Table 8 for the Op-Mux module. The default configuration is A-OP-B, which makes the Op-Mux module transparent on the datapath by directly connecting A to X and B to Y. The output port Y is used to implement the folding. If one of the fold configurations is selected, both output ports X and Y receive a combination of the register file's port A bit streams. As port B of the register file is not used in this configuration, we can use this port to simultaneously write the ALU output to the destination registers. This optimization reduces the cycle latency in (4) even further. If we assume there exist C (= 1, 2, 4) pipeline stages in the datapath, the equation for the reduction latency using the Op-Mux module becomes,

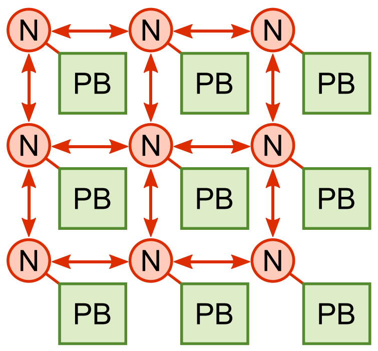
$$\text{PB-RL}_{\text{opmux}} = (N + C) \cdot \log_2 q \tag{5}$$

In a PE-Block with q = 16 and N = 32, PB-RL$_{\text{NEWS}}$ takes 736 cycles. Even with the deepest pipeline (C = 4) PB-RL$_{\text{opmux}}$ takes 144 cycles to perform a PE-Block level accumulation, providing a 5.1× improvement. In our experimental design, the muxing configurations of Table 8 are implemented for 16-bit wide ports A, B, NET, X, and Y using 20 LUTs and 3 flip-flops on a Virtex-7 FPGA. The flip-flops are used for storing the configuration code. The combinatorial part acts as a single LUT stage in the datapath.

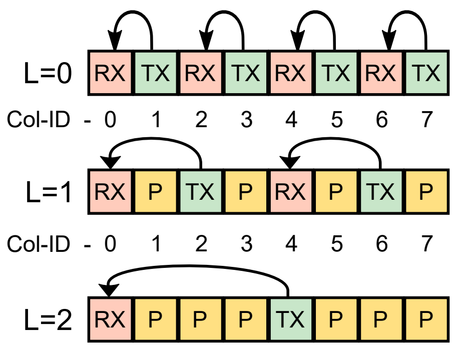
4.1.4 Data Network Architecture

The proposed network architecture uses a PE-Block instead of a PE as a node, unlike a standard NEWS network design [29]. As shown in Fig. 6(a) a PE-Block connects to the network module, which in turn connects to the network. Though the figure shows a typical NEWS network, any other network configuration (e.g. mesh network) between the nodes (N) can be implemented. We can assume that the PE-Block level accumulation result is stored in a PE-0 register of all PE-Blocks. To get the accumulation result of the entire row of the array, we need to accumulate only those registers.

Fig. 6(b) shows the data movement pattern for reduction across the array processor. A network module can be abstracted as a flip-flop with some routing logic. A conceptual binary tree is implemented using a hopping mechanism along a row or a column. We use a conceptual level



(a) Network Architecture



(b) Jump over PE-Blocks

Figure 6: Data network for fast accumulation and reduction operations

value (L) to encode the routing configuration of each node. A node can be configured in 3 modes: transmitter (TX), receiver (RX), and pass-through (P). Each node has a unique row and a unique column ID corresponding to its position in the array. Based on the level specified and the unique IDs, a node operates in one of these three modes.

In Fig. 6, the movement pattern for level 0 shows that adjacent nodes get logically connected because every even column acts as a receiver while every odd column acts as a transmitter. As each node adds a register stage, level 0 has a network latency of 1 hop. Similarly, for level 1, in every consecutive 4 nodes, the first one acts as the receiver, the second and the last ones act as pass-through, and the third one acts as the transmitter. This effectively connects the first and the third of them, with a network latency of 2 hops. Finally, level 2 connects node-4 to node-0 with a network latency of 4 hops. In general, the number of hops (H) needed for a given level can be represented by (6). At the end of level 2, PE 0 of node-0 contains the accumulation result of the entire row of the array. We can say that each level of the conceptual binary tree facilitates a jump over the intermediate PE-Blocks. In a network with D nodes in a row, the number of such jumps (J) needed can be expressed using (7).

$$H = 2^{\mathrm{L}} \tag{6}$$

$$J = \log_2 D \tag{7}$$

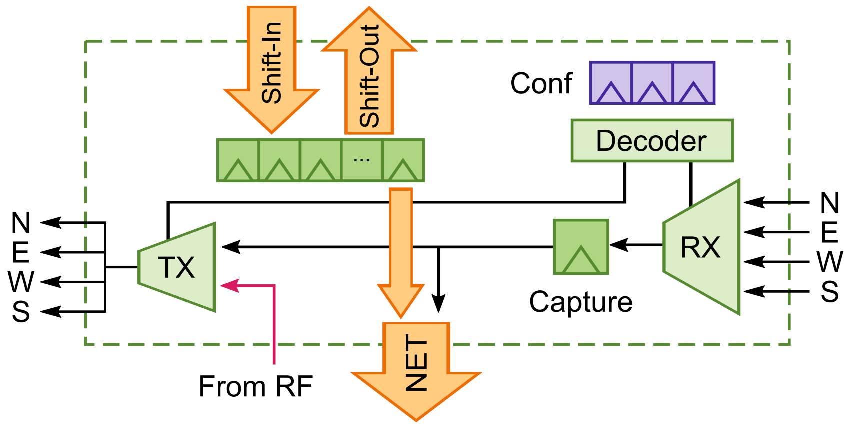


Figure 7: Architecture of the network node for binary hopping

4.1.5 Network Node Architecture

Fig. 7 shows the architecture of the network module that acts as a node (N) in the data network. It has 3 logical parts: receiver multiplexer (RX), transmitter multiplexer (TX), and capture registers. It also has a configuration register that holds the current level value of the conceptual tree and the direction of the data movement. The level value is decoded into the selection bits of the TX mux, and the direction value determines the selection bits of the RX mux. A single flip-flop is enough to capture the network stream of bit-serial PEs for accumulation along the row. Custom PIM blocks can have a single-bit port connecting the output of the capture register to the multiplexers at the SA output [1,99]. Accumulation operation is not essential along the columns in deep-learning applications. However, for convolution and pooling operations in CNNs, each PE needs to access at least its adjacent PE. This can be enabled using a set of flip-flops acting as a shift register along each column of a PE-Block.

In the proposed design of PiCaSO, we implement the exact architecture shown in Fig. 7. Because the dimension of the PE-Block is 1×16, we added 15 flip-flops for the shift along the column. The RX mux is implemented using a LUT6 and TX mux is implemented using a LUT3 primitive. The conceptual binary-tree level encoding makes the network node design highly scalable. We selected 3 bits to encode the level in each direction, which can be decoded by a LUT6. Now, a 3-bit number can be used to specify 2^3 or 8 levels. A network with 8 levels can

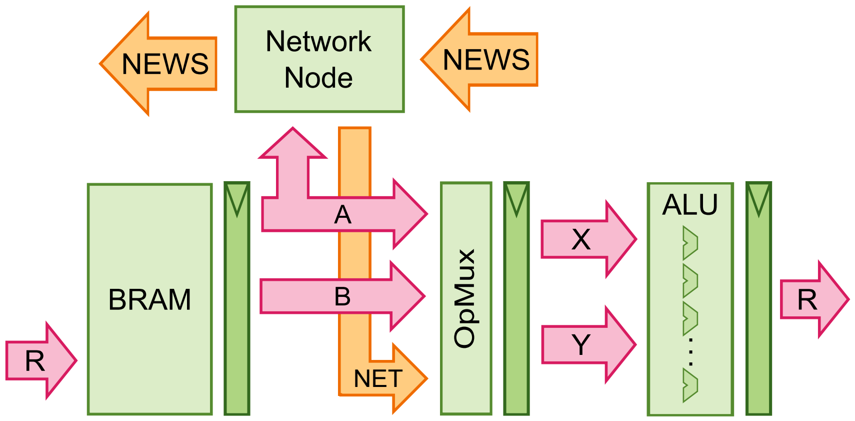


Figure 8: Proposed PIM Overlay with all stages pipelined (Full-Pipe)

handle a 256×256 array of PE-Blocks. As each PE-Block has 16 PEs, such an array, if implemented, will contain more than a million PEs.

4.1.6 Datapath Pipeline

As shown in Fig. 8, there are 3 stages in the datapath, which can be pipelined to increase the clock frequency: register file output, Op-Mux output, and ALU output. The datapath pipeline needs to be designed carefully, especially for bit-serial PE architectures. If an extra clock cycle is introduced in a processing step of a bit-serial PE, the operation latency will scale linearly with the size of the operand for basic operations like ADD/SUB and quadratically for operations like MULT. We studied the performance and utilization of different pipeline configurations, which are presented in Table 9.

Single-Cycle: In this configuration, none of the pipeline stages shown in Fig. 8 are enabled. This corresponds to the traditional RF–ALU–RF datapath presented in prior work [29] as well as the modified BRAM block architectures in [1, 99]. A single cycle is needed to read a bit of the operands from the register file, process it through the ALU, and send the result back to the register file data port to be written in the next cycle. The controller FSM repeatedly transitions between Read–Write states to compute the result in 2N cycles for N-bit operands. This configuration has the slowest datapath, as expected. Our experimental study shows that about

Table 9: Comparison of different PE-Block configurations in Virtex-7

	Prev. Work [29]	Full-Pipe	Single-Cycle	RF-Pipe	Op-Pipe
LUT	187	53	82	74	53
FF	64	119	71	103	103
Slice	64	32	42	36	37
Max-Freq	270 MHz	540 MHz	265 MHz	400 MHz	390 MHz

80% of the total delay is contributed by the register file.

RF-Pipe: We considered the effects of adding a pipeline stage at the output of the register file. As shown in Table 9, this bumps up the clock frequency to 400 MHz, 1.5$\times$ improvement compared to the Single-Cycle configuration. In this configuration, 2 clock cycles are needed to read a single bit out of the register file. This overhead can be remedied by modifying the controller FSM to transition through 2$\times$Read–2$\times$Write states. As a result, it takes 4 cycles to read, process, and write 2 bits of the operand, which takes 2N cycles to process N-bit operands. The implied condition is that N is an even number, which is almost always the case.

Op-Pipe: We next considered adding a pipeline stage at the output of Op-Mux. This configuration achieves 390 MHz, which is very close to the performance achieved with the RF-Pipe configuration. The difference is due to the longer output delay of the BRAM in the absence of its output registers. This configuration also requires 2 clock cycles to process a single bit of the operand, which can be handled in a similar manner as in the RF-Pipe configuration. The unique advantage of this configuration becomes more evident when the PE Block is connected to a larger network. The pipeline stage at the output of Op-Mux can help the design tools to meet the timing of the paths that pass through the network, which may have long wire delays.

Full-Pipe: We then considered fully pipelining the stages as shown in Fig. 8. This configuration requires 4 clock cycles to read a single bit out of the register file, process it through the ALU, and send the result back to the register file to be written in the next cycle. This overhead can be remedied by extending the solution for RF-Pipe: the controller FSM to transition through

4×Read – 4×Write states. To process N-bit operands, it still takes 2N clock cycles. The implied condition, in this case, is that N is a multiple of 4 (e.g. 4, 8, 16, etc.), which is the most common case. As shown in Table 9, this configuration achieves a clock frequency of 540 MHz, which is 2× that of the Single-Cycle configuration shown in Fig. 3. This implementation targets Xilinx's xc7vx485tffg1761-2 device, which is a Virtex-7 FPGA of -2 speed grade. The maximum frequency of BRAM in this device reported in the datasheet is 543.77 MHz. Thus, this configuration meets one of the proposed Standards of an Ideal PIM design: clock at the BRAM Fmax.

4.1.7 Accumulation Latency

Suppose, (a) we have a PE array of dimension $q \times q$, (b) operating on N-bit operands, (c) built from the Full-Pipe configuration ($C = 4$) of the PE-Block as a 1×16 array of PEs, (d) connected to the network shown in Fig. 6 through the node shown in Fig. 7. From (5) we can compute PE-Block level accumulation latency as,

$$\begin{aligned} \text{PB-RL}_{\text{opmux'}} &= (N+4) \cdot \log_2 16 \\ &= 4N + 16 \end{aligned} \tag{8}$$

From (7) we observe $J = \log_2(q/16)$ jumps are needed, from level 0 to level $J-1$, in order. Using (6), the total hop latency to perform all jumps over q PEs can be computed as,

$$\begin{aligned} H_q &= 2^J - 1 \\ &= \frac{q}{16} - 1 \end{aligned} \tag{9}$$

The last part of the accumulation is to stream the output of the network capture register through the ALU and ADD it to a operand stream in the receiver PE-Block. In each jump, it takes 4 cycles to fill the pipeline and N cycles to write the result back to the register file. Thus, the operate and write-back latency (WB) for all jumps can be expressed as,

$$\mathrm{WB}_q = (N+4)\cdot J \tag{10}$$

Note that in the Full-Pipe configuration, 2 cycles are needed in the transmitter PE-Block to get the first bit of the stream to enter the network before hopping starts. This latency is included in (10). From (8), (9), and (10) we compute the total reduction latency at the array level ($\text{Arr-RL}_{\text{Full-Pipe}}$) as follows,

$$\begin{aligned}\text{Arr-RL}_{\text{Full-Pipe}} &= \text{PB-RL}_{\text{opmux'}} + H_q + \mathrm{WB}_q \\ &= 4N + 16 + \frac{q}{16} - 1 + (N+4)\cdot J \\ &= 15 + \frac{q}{16} + 4N + (N+4)\cdot J \end{aligned} \tag{11}$$

As observed in (11), the array dimension contributes two terms: $q/16$ and $(N+4)\cdot J$, both of which grow significantly slowly with the array size, q, making the network design highly scalable.

We can rewrite (11) in the form of the proposed ideal latency model (1). In this architecture, a partial sum is generated from each PIM block. Thus, P $= q/16$ and $J = \log_2(q/16) = \log_2(P)$. Substituing these values we can write,

$$\begin{aligned}\text{Arr-RL}_{\text{Full-Pipe}} &= 15 + P + 4N + (N+4)\cdot\log_2(P) \\ &= (N+4)\cdot\log_2(P) + 1\cdot P + (15+4N)\end{aligned} \tag{12}$$

Comparing the array reduction latency form (12) with the ideal latency model (1) shows that the implementation specific parameters for this architecture are, $a = (N+4)$, $b = 1$, $c = (15+4N)$. These values are in the ideal range of the proposed latency model as shown in Table 4. Thus, PiCaSO accumulation network architecture meets the ideal latency standard.

4.2 Analysis

In this section, we will study the utilization and performance of FPGA implementations of PiCaSO. To highlight its relative improvements, we have compared PiCaSO against a benchmark PIM overlay architecture [29]. A scalability study has been presented, demonstrating that PiCaSO adheres to the ideal scalability standard. Additionally, we have performed a detailed comparative analysis with custom-BRAM PIM architectures, exploring the advantages and disadvantages of the PiCaSO architecture in contrast to those designs. This comprehensive examination provides insights into PiCaSO's strengths and unique architectural features.

4.2.1 Performance and Utilization

Table 10 compares the pipeline configurations outlined in subsection 4.1.6 against SPAR-2, the benchmark overlay from [29]. All designs were implemented and run on Virtex-7 (xc7vx485) and Alveo U55 FPGAs. Utilization numbers follow the tile definition in SPAR-2 consisting of 256 PEs organized in a 4×4 array of PE blocks, with 16 PEs in each block. The total utilization per tile and the average utilization per block are shown. The Full-Pipe configuration achieved a 2.25× and a 1.67× increase in clock frequency compared to the benchmark design on Virtex-7 and U55 devices, respectively. In both devices, Full-Pipe provided a 2× improvement in resource utilization over SPAR-2.

The Single-Cycle configuration achieved similar performance on the Virtex-7 and better performance on the U55 compared to the benchmark system, with 2.6× and 2.5× utilization improvements, respectively. It had a smaller flip-flop count and slice utilization compared to the Full-Pipe due to the absence of the pipeline registers. Both RF-Pipe and Op-Pipe achieved better clock speeds but with an increase in slice utilization compared to Single-Cycle, due to the addition of the pipeline stages. As argued in Subsection 4.1.6, Op-Pipe had better performance compared with RF-Pipe by minimizing the clock latency contributed by the network. All configurations offered at least 2× better utilization and up to 2× better performance compared to the benchmark design.

Table 10: Performance and Utilization of Comparison between PIM Blocks (B) and 4 $\times$ 4 Tiles (T) of different pipeline configurations

	Benchmark [29]				Full-Pipe				Single-Cycle				RF-Pipe				Op-Pipe			
	Virtex-7		U55		Virtex-7		U55		Virtex-7		U55		Virtex-7		U55		Virtex-7		U55	
	T	B	T	B	T	B	T	B	T	B	T	B	T	B	T	B	T	B	T	B
LUT	3023	189	2449	153	835	52	774	48	895	56	1068	67	1017	64	1064	67	836	52	774	48
FF	1024	64	768	48	1799	112	1799	112	1031	64	1031	64	1543	96	1527	95	1543	96	1543	96
Slice	1056	66	556	35	522	33	243	15	395	25	223	14	451	28	243	15	472	30	295	18
Max-Freq	240 MHz		445 MHz		540 MHz		737 MHz		245 MHz		487 MHz		360 MHz		600 MHz		370 MHz		620 MHz	

Table 11: Cycle latency of different operations

Operation	Benchmark [29]	PiCaSO-F
ADD/SUB	$2N$	$2N$
MULT[1]	$2N^2+2N$	$2N^2+2N$
Accumulation[2]	$(q-1+2\log_2 q)N$	$15+\frac{q}{16}+4N+(N+4)J$
q = 128, N = 32	4512	259

[1] Booth's Radix-2 multiplication
[2] q : Number of columns to be accumulated
N : Operand width
J : Number of network jumps needed = $\log_2(q/16)$

Table 10 shows Full-Pipe achieved clock frequencies of 540 MHz on the Virtex-7 (xc7vx485-2), and 737 MHz on the Alveo U55 (xcu55c, -2 speed grade). The data sheets for these devices list 543.77 MHz and 737 MHz, respectively as the maximum BRAM clock frequencies. Surprisingly, this is an improvement over the custom designs reported in [1, 99]. The technology node of Alveo U55 (16 nm) is comparable to that of the designs proposed in CCB (Stratix 10, 14 nm) and CoMeFa (Arria 10, 20 nm). Yet, PiCaSO runs 1.62× and 1.25× faster than the fastest configurations of CCB and CoMeFa, respectively. This is due to the pipelined architecture of PiCaSO, where the slowest stage is the BRAM. Thus, it can run as fast as the maximum frequency of the BRAM, achieving the ideal clocking standard.

4.2.2 Reduction Network

Both PiCaSO and SPAR-2 [29] use Booth's Radix-2 algorithm for multiplication. Thus, the cycle latencies for the ADD/SUB and MULT operations in Table 11 are identical. SPAR-2 uses a standard NEWS network to copy operands between PEs when summing the partial products during multiply-accumulate (MAC) operations. The Accumulation row compares the number of clock cycles for SPAR-2's NEWS network and PiCaSO's reduction network. The last row in Table 11 shows the PiCaSO-F reduction network provides a 17× improvement in accumulation latency for the test configuration reported in [25]. This improvement is due to the careful design of the binary-hopping network discussed in Section 4.1.4, which overlaps data transfer with

Table 12: Representative of Virtex-7 and Ultrascale+ devices

Device	Tech	BRAM#	Ratio[1]	Max PE#[2]	ID
xc7vx330tffg-2	V7	750	272	24K	V7-a
xc7vx485tffg-2	V7	1030	295	32K	V7-b
xc7v2000tfhg-2	V7	1292	946	41K	V7-c
xc7vx1140tflg-2	V7	1880	379	60K	V7-d
xcvu3p-ffvc-3	US+	720	547	23K	US-a
xcvu23p-vsva-3	US+	2112	488	67K	US-b
xcvu19p-fsvb-2	US+	2160	1892	69K	US-c
xcvu29p-figd-3	US+	2688	643	86K	US-d

[1] LUT-to-BRAM ratio
[2] Maximum number of PEs if all BRAMs are utilized

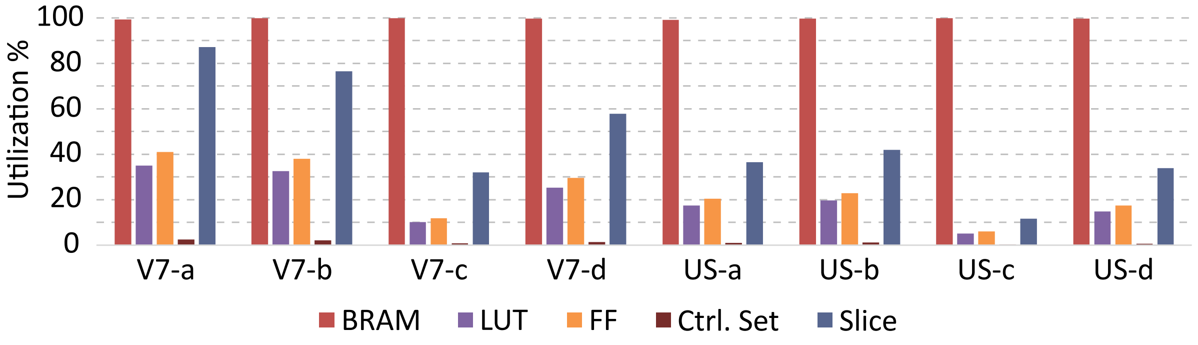


Figure 9: Scalability study on Virtex-7 and Ultrascale+ FPGA families

computation during accumulation.

4.2.3 Scalability

PiCaSO is designed to meet the ideal scaling goal: linear scaling of peak-performance with the BRAM capacity of the target device. To evaluate scalability, the largest-sized array of PIM blocks that could fit into the target devices was constructed. The results of this study are shown in Table 13.

In the Virtex-7 FPGA, the largest array of SPAR-2 [29] PIM blocks contained 24K PEs. This did not achieve the full capacity of the Slices or BRAM resources available in that device. The implementation tool failed at the placement step for larger arrays due to a high utilization (32.1%) of Unique Control Sets. Control sets are the collection of control signals for slice flip-flops [100]. Flip-flops must belong to the same control set to be packed into the same slice. A large number of

Table 13: Comparison of largest overlay arrays in Virtex devices

	Virtex-7		Alveo U55	
	Benchmark [29]	PiCaSO-F	Benchmark [29]	PiCaSO-F
Max-Size	24K	33K	63K	64K
LUT	74.6%	32.5%	41.6%	14.8%
FF	16.0%	38.0%	9.7%	17.3%
BRAM	73.8%	99.9%	98.4%	100.0%
Uniq. Ctrl. Set	32.1%	2.1%	19.5%	0.8%
Slice	86.0%	76.4%	63.4%	32.0%

unique control sets makes it difficult to find a valid placement, even with enough available slices. In contrast, PiCaSO-F fully utilized the BRAM resources to fit 33K PEs, a 37.5% improvement over SPAR-2 in the same device. PiCaSO does not suffer from the placement issues observed in SPAR-2 due to a very low (2.1%) utilization of the unique control sets.

In the U55 FPGA, SPAR-2 almost achieved the full BRAM capacity for an array size of 63K PEs. This is due to the U55 FPGA offering significantly more slices and routing resources compared to the Virtex-7 FPGA. PiCaSO achieved 100% utilization of BRAM with 2$\times$ better slice utilization over SPAR-2.

Our results showed that the scalability of the benchmark design, SPAR-2, is dependent on the Slice-to-BRAM ratio and cannot guarantee the creation of a PIM array that scales with the BRAM capacity. Conversely, our results showed PiCaSO scaling with the BRAM capacity independent of the Slice-to-BRAM ratio across multiple devices of Virtex-7 and Ultrascale+ FPGA families. Table 17 lists representative devices we evaluated based on the following two criteria: BRAM capacity and LUT-to-BRAM ratio. Each device is assigned an ID as a short name to be used in this paper.

Fig. 9 shows that PiCaSO utilized the full BRAM capacity in all devices, and achieved the maximum number of PEs the device can fit based on BRAM density. Results showed for the smallest device (V7-a) and lowest LUT-to-BRAM ratio, the LUT and flip-flop utilization is around 40%. For one of the largest devices with a high LUT-to-BRAM ratio (US-c), these utilization numbers are negligible, around 5%. These results strongly support that PiCaSO scales linearly with the BRAM capacity of the device.

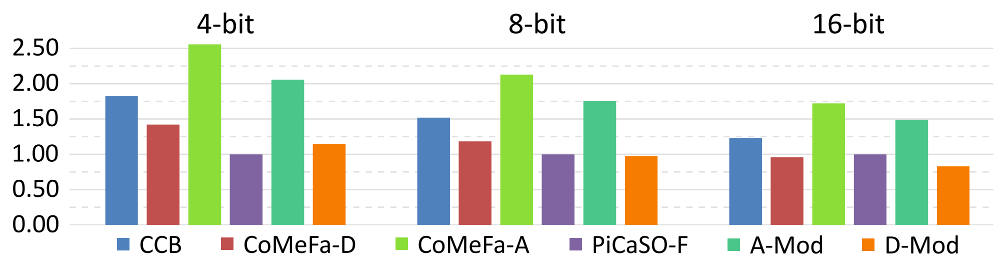


Figure 10: Relative MAC latency of custom designs w.r.t PiCaSO

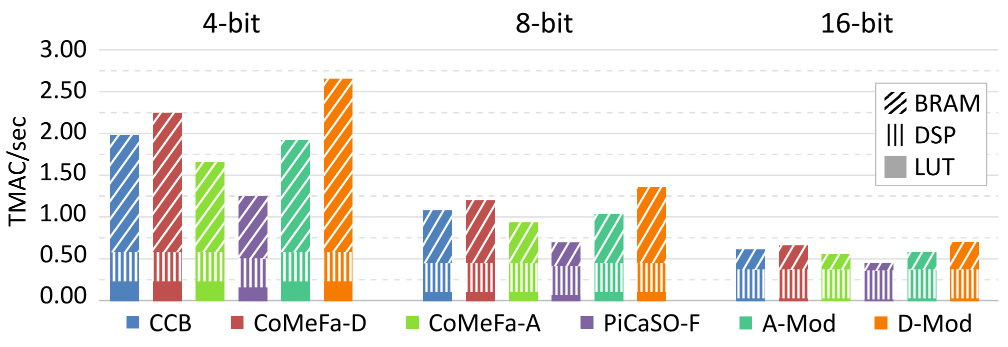


Figure 11: Peak MAC throughput of PiCaSO and custom designs on Alveo U55

4.3 Comparison with Custom-BRAM PIM Designs

Fig. 10 shows the relative MAC latency of the custom designs w.r.t PiCaSO. The latency is computed for 16 parallel MULTs followed by the accumulation of the products. The clock speeds of custom designs are adjusted based on the performance degradations reported in [1, 99]. With the exception of CoMeFa-D at 16-bit precision, PiCaSO has the shortest latency due to faster clock speed and accumulation. CCB and CoMeFa extend the clock period to allow a complete read-modify-write per clock cycle. This allows a complete MULT to finish in half the number of cycles compared to PiCaSO and can reduce latencies at higher precisions. Still, PiCaSO runs $1.72\times$ – $2.56\times$ faster than CoMeFa-A, which is reported as the most practical design in [1].

Peak TeraMAC/sec throughputs on the U55 FPGA are shown in Fig. 11. CCB and CoMeFa design the BRAM IP to support one PE per bitline. With a column muxing factor of 4 [1], a Virtex 36Kb BRAM would be redesigned as a 256×144 array with 144 PEs per BRAM. The use of standard BRAM IP prevents PiCaSO (and all overlays) from making this modification. Yet PiCaSO still achieves 75% – 80% of CoMeFa-A's peak throughput, the most practical of the two

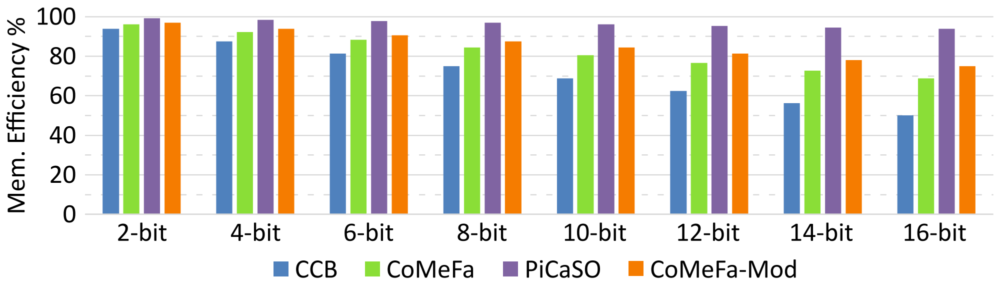


Figure 12: BRAM memory utilization efficiency on Virtex devices

CoMeFa designs. This results from PiCaSO not sacrificing the same degradation of clock speed seen in all custom designs.

The memory utilization efficiency of BRAMs is not discussed in [1, 99] but we feel is an important metric for PIM architectures. Memory utilization efficiency can be defined as the fraction of BRAM memory that can be used to store model weights. Both CCB and CoMeFa follow the computation techniques used in [90] which requires scratchpad memory. For N-bit operands, CCB requires 8N reserved wordlines. CoMeFa only needs 5N wordlines using the "One Operand Outside RAM (OOOR)" technique. PiCaSO requires only 4N wordlines, as it does not require copying operands to the same bitline as in CoMeFa. In the widest mode of a Virtex 36Kb BRAM, each PE of CCB and CoMeFa would have 256 bits of storage in its register file (bitline). For PiCaSO, each register file has 1024 bits. Fig. 12 shows the memory utilization efficiency of these architectures. As observed, at higher precisions the memory efficiency drops significantly for CCB and CoMeFa. For 16-bit operands, CCB and CoMeFa are only 50% and 68.8% efficient, respectively, while PiCaSO is 93.8% efficient.

Table 14 summarizes comparisons between PiCaSO and the custom designs. The custom designs significantly degrade the BRAMs maximum clock frequency, whereas PiCaSO runs at the maximum clock speed of the BRAM. However, PiCaSO has 1/4th the number of parallel MACs, as it cannot access all the bitlines. Multiplication in PiCaSO is 2$\times$ slower, as it requires 2 cycles to process a single bit. However, accumulation is 2$\times$ faster in PiCaSO. PiCaSO supports Booth's radix-2 multiplication. In Booth's algorithm, on average half of the intermediate steps are NOPs. Thus, PiCaSO can potentially further reduce the multiplication latency by 50% on average.

Table 14: Comparison with Customized BRAM PIM architectures

	CCB	CoMeFa-D	CoMeFa-A	PiCaSO-F	A-Mod
Architecture	Custom	Custom	Custom	Overlay	Custom
Clock Overhead	60%	25%	150%	0%	150%
Parallel MACs	144	144	144	36	144
Mult Latency[1]	(a)	(a)	(a)	(b)	(a)
N = 8	86	86	86	144	86
Accum. Latency[2]	(c)	(c)	(c)	(d)	(e)
q = 16, N = 8	80	80	80	48	40
Support Booth's	No	Partial	Partial	Yes	Yes
Mem. Efficiency	Low	Medium	Medium	High	Medium
Complexity	High	Medium	Medium	No	Medium
Practicality	Low	Medium	High	Very High	High

[1] (a) N^2+3N-2 ; (b) $2N^2+2N$

[2] (c) $(2N+\log_2 q)\log_2 q$; (d) $(N+4)\log_2 q$; (e) $(N+2)\log_2 q$

CoMeFa can use Booth's algorithm only in OOOR mode and CCB does not support it at all.

As discussed earlier, the memory utilization efficiency of CCB is significantly low, PiCaSO is high, and CoMeFa lies in between. CCB has the highest design complexity mainly due to its need for a modified voltage supply. CoMeFa has medium complexity since it requires modifications to the SAs, additional flip-flops, and SA cycling. Being an overlay, PiCaSO does not have such design complexities. As reported in [1], the practicality of CCB is low, CoMeFa-D is medium, and CoMeFa-A is high. In that reference, the practicality of PiCaSO is very high. It offers 80% of CoMeFa-A's peak throughput with 2.56× shorter latency, 25% better memory efficiency, can be implemented using off-the-shelf FPGAs, and is tested on real devices, while CCB and CoMeFa numbers are mainly based on simulations.

4.4 Fusing PiCaSO Optimizations into Custom-BRAM Designs

Fig. 13 shows how modifications highlighted in red can accelerate CoMeFa-A [1]. We refer to this implementation as *A-Mod*. PiCaSO's OpMux module per bitline consists of a 2-to-1 mux and a 4-to-1 mux. This can be implemented using a few CMOS pass transistors. OpMux then saves both the cycles and memory needed to copy operands during accumulation [90, 99]. PiCaSO's

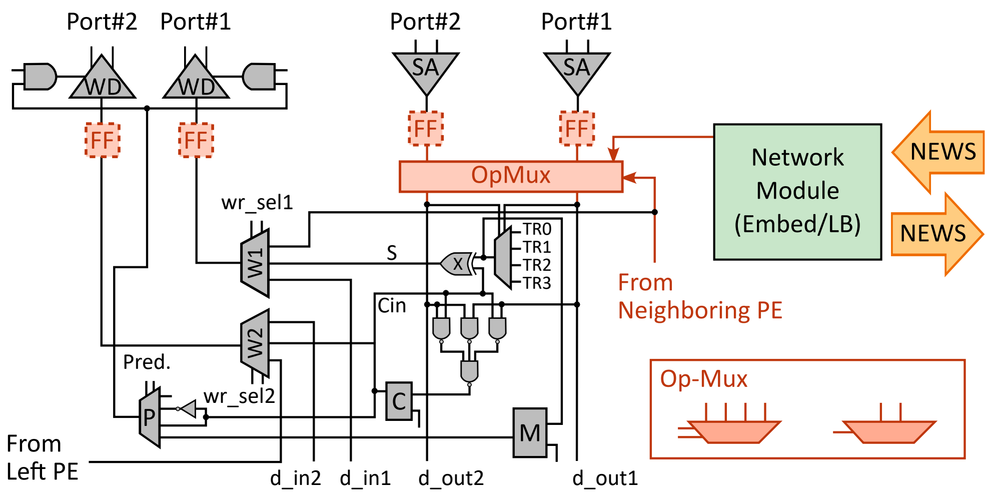


Figure 13: Modified CoMeFa-A [1] with PiCaSO adoption (A-Mod)

network module can overlap data movement with computation between different PIM blocks. The network module can be embedded within the PIM block or can be implemented using logic slices from the FPGA. A single-bit port connection to the network module is enough to support row-wise accumulation.

Although [1] mentions that the PE does not add additional delay to the extended clock, in a practical circuit, there will always be an additional delay. This delay can be hidden using one of the pipelining schemes of PiCaSO. A single stage of registers could be enough to hide the PE delay. As BRAM blocks already contain output registers, this should not add any additional area overhead on top of what is reported in [1]. The PE circuit can be placed between two stages of registers if the delay is too long. This is illustrated using the dashed flip-flops in Fig. 13. Similar modifications can be performed on CoMeFa-D referred to as implementation *D-Mod*.

These modifications can significantly improve the performance of the custom designs. The extrapolated performance numbers for A-Mod and D-Mod are presented in Fig. 10 and Fig. 11. As observed in Fig. 10, the adoption of PiCaSO's OpMux and network modules can improve their MAC latency by 13.4% – 19.5% due to faster accumulation. This consequently improves their throughput by 5% - 18% over different precisions. In Fig. 12, CoMeFa-Mod represents both A-Mod and D-Mod implementations. Due to OpMux, A-Mod and D-Mod no longer requires scratchpad storage to copy operands for accumulation. This improves their memory utilization efficiency by 6.2%. This means at 4-bit precision, 1.6 million more weights can be stored in a device with 100 Mb of BRAM. This would significantly reduce weight stall cycles [38] and allow bigger models to be stored on chip. In Table 14, the A-Mod column shows the architectural enhancements due to these modifications. A-Mod retains the high parallelism and fast Mult latency of the original CoMeFa design and offers 2× faster accumulation and full support for Booth's algorithm.

4.5 Discussion

All these results support the practicality of the proposed ideal design standard for PIM array accelerators. Although PiCaSO by itself is not a complete accelerator system, its ability to operate at the BRAM Fmax and scale with the BRAM capacity makes it relatively easier to build an accelerator that possesses similar attributes. Furthermore, because PiCaSO's reduction network already adheres to the ideal standard, it will significantly reduce the design complexity of the data network at the system level. This adherence simplifies the task of maintaining compliance with the scalability standard, ensuring that any system built using PiCaSO can achieve high performance and efficient resource utilization. Thus, PiCaSO serves as a robust foundation for developing high-performance and scalable accelerators.

Chapter 5

IMAGine: An In-Memory Accelerated GEMV Engine

While the Ideal Design Standard is an ultimate objective to strive for, it can also serve as a framework for crafting an efficient architecture of a PIM-Array accelerator on FPGAs. In this chapter, we develop IMAGine, an In-Memory Accelerated GEMV engine, using PiCaSO. IMAGine is implemented and analyzed on an off-the-shelf FPGA. The development of IMAGine serves a dual purpose: it provides a case study demonstrating the use of the proposed Standard to make better design choices at the system level and evaluate how achievable those standards are in a practical accelerator. IMAGine has been published at [101] as an open-source implementation and is freely available for study, use, modification, and distribution without restriction.

5.1 Architecture

5.1.1 System-Level Architecture

The top-level system is illustrated in Fig. 14(a). It consists of (1) a 2D array of GEMV tiles, (2) a set of input registers, (3) a fanout tree connecting the input registers to the tile array, and (4) a column of shift-registers to read out the final result. The input registers are used by the front-end processor to send instructions to the GEMV tile controllers. The fanout tree is parameterized to be adjusted during implementation. The 2D tile array is implemented as a parameterized module that instantiates and connects GEMV tiles to build the tile array. The bits written to the register file of the leftmost PE in the array are shifted into the column shift registers. At the end of the GEMV operation, the output vector is stored in the column shift registers, which can be shifted up and read through the FIFO-out port, one element per cycle.

5.1.2 IMAGine Tile Architecture

Illustrated in Fig. 14(b), the GEMV tile is the heart of IMAGine. It consists of (1) an FSM-based controller, (2) a 2D array of PIM blocks, and (3) a fanout tree between them. The controller

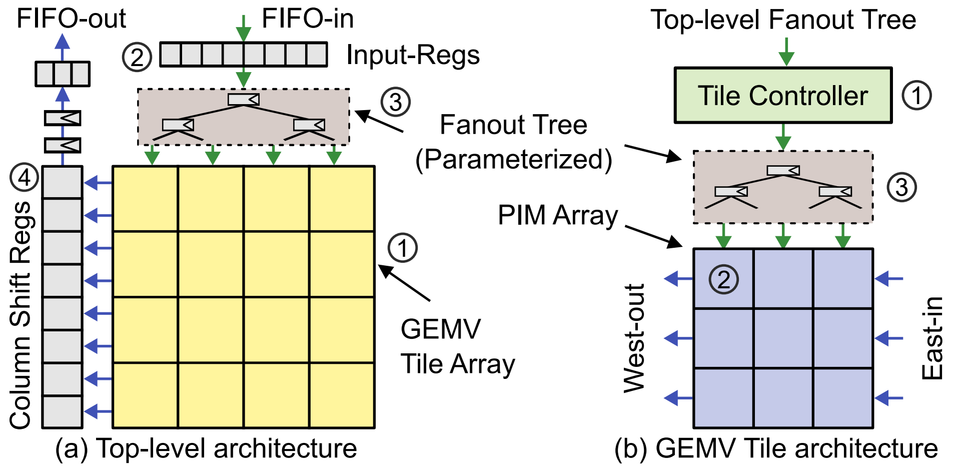


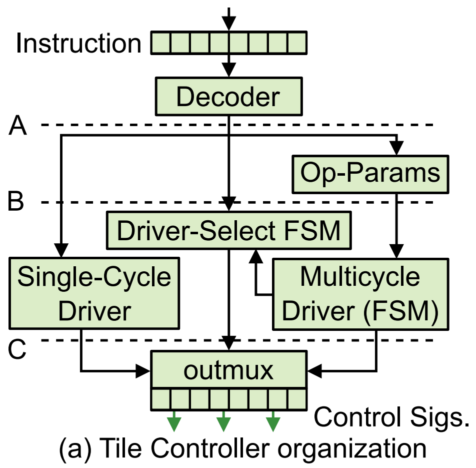
Figure 14: System architecture of IMAGine illustrating the data and instruction flow (a) through the GEMV engine and (b) within GEMV tiles.

receives the instruction written to the input registers at the top level, decodes it, and generates the sequence of control signals needed to execute the instruction. The fanout tree connects the control signals to all PEs in the PIM array and is parameterized for adjustment during implementation. The PIM array interfaces allow cascading with arrays in neighboring tiles on each side. During accumulation, partial results move from east to west through PIM arrays, ultimately accumulating in the left-most PE column of the left-most GEMV tile in a row.

5.1.3 Tile Controller

Fig. 15(a) shows the architecture of the tile controller. As discussed in Section 3.2, logic paths need to be short enough to achieve the ideal clock rate. However, estimating precise logic depth during RTL design is challenging and the requirement varies across devices. Thus, we grouped the combinatorial logic into meaningful steps and added optional pipeline stages illustrated by the dashed lines A, B, and C in Fig. 15(a). Running synthesis, we ensured that each step could be implemented in one or two logic levels.

The controller takes a 30-bit instruction, which is executed by either the single-cycle or the multicycle driver. The 2-state driver-selection FSM enables any one of them based on the opcode. The single-cycle driver can execute one instruction every cycle, while the multicycle driver takes several cycles to execute instructions like ADD, SUB, MULT, etc. including an additional cycle



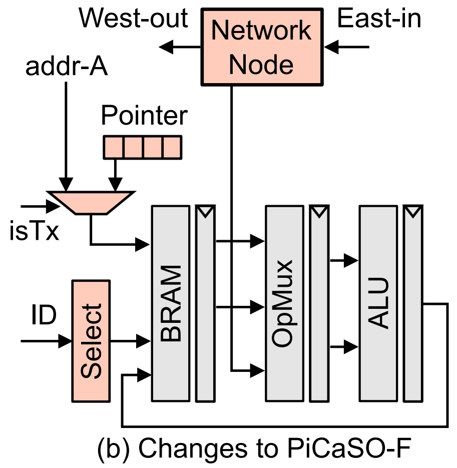


Figure 15: Architectures of (a) GEMV controller and (b) PiCaSO-IM, the adapted version of PiCaSO-F [2].

to load its parameters from the Op-Params module. All inputs and outputs are registered to localize timing paths within the controller.

5.1.4 PIM Module

PiCaSO-F, the fully pipelined configuration of PiCaSO, is used to build IMAGine. Some modifications to PiCaSO-F were needed to enhance its control capabilities to implement the GEMV tile. These modifications are highlighted in red in Fig. 15(b). The additional logic supporting the original NEWS network was removed, keeping only the parts needed for east-to-west data movement. PiCaSO-F lacked control signals for selectively enabling/disabling a block required in IMAGine. Block-ID-based selection logic was included in PiCaSO-IM. Our accumulation algorithm needed 3 addresses to maximize the overlap of data movement and computation. As PiCaSO-F supported only 2 simultaneous addresses, we added a pointer register for the third address.

If PiCaSO is realized as a custom-BRAM tile as discussed in Section 4.4, these changes can be implemented in programmable logic fabric, keeping registerfile, OpMux, and ALU modules within the BRAM tile. We name such a custom-BRAM implementation of PiCaSO-IM as PiCaSO-CB.

Table 15: Utilization and Clock Frequency of Modified PiCaSO 4×4 Tile

	LUT	FF	Slice	DSP	BRAM	Max Freq.
PiCaSO-F Tile	774	1799	243	0	8	737 MHz
PiCaSO-F Block	49	113	15	0	0.5	737 MHz
PiCaSO-IM Tile	1352	1989	264	0	8	737 MHz
PiCaSO-IM Block	85	125	18	0	0.5	737 MHz
Change %	74.7%	10.6%	8.6%	-	0.0%	0%

5.2 IMAGine Implementation and Analysis

In this section, we discuss the bottom-up implementation of IMAGine, setting the design goal to be the Ideal Standard discussed in Chapter 3. IMAGine was studied on AMD Alveo U55C (xcu55c, -2 speed grade). The BRAM Fmax on this device is 737 MHz [86], which sets the target clock period to be 1.356 ns. All of the following studies were carried out using Vivado 2022.2.

5.2.1 PiCaSO-IM Block

We first verified that the additional logic added to the original PiCaSO-F did not degrade the BRAM Fmax within the PIM block or create a logic utilization bottleneck. A 4×4 array of the new PiCaSO-IM was tested and compared with PiCaSO-F. This comparison is shown in Table 15. The Change% row shows the utilization and clock speed change compared to the original implementation. BRAM utilization is 0.5 because a PiCaSO block uses one RAMB18 tile, which is reported by Vivado as 0.5 number of RAMB36 tile of AMD devices. As observed, the modifications did not affect the clock frequency and the utilizations of BRAM and DSP. The flip-flop utilization increased by only 10.6%. Though there is a significant increase (74.7%) in the LUT utilization, the overall Slice utilization only increased by 8.6%. This means that the additional logic has a high packing factor in the logic slices. As a result, the additional logic would not be expected to introduce a utilization bottleneck.

Table 16: Utilization and Frequency of 12×2 GEMV Tile Components

	Controller	Rel.	Fanout	Rel.	PIM Array	Rel.	Tile
LUT	167	5.8%	0	0.0%	2736	94.2%	2903
FF	155	4.0%	615	15.9%	3096	80.1%	3866
DSP	0	-	0	-	0	-	0
BRAM	0	0.0%	0	0.0%	12.0	100.0%	12
Freq. (MHz)	890	1.2×	890	1.2×	737	1×	737

5.2.2 IMAGine GEMV Tile

Before we implemented the GEMV tile discussed in Section 5.1.2, its components were studied individually to verify if they met the design requirements. The GEMV tile contains a 12×2 PIM array and 2 stages of pipeline in the fanout tree. This configuration best fits the physical layout of the Alveo U55 FPGA as discussed later in this section. Table 16 shows the utilization and performance of these components and their relative values compared to the entire GEMV tile.

The controller together with the fanout network passed the timing constraints at a clock rate up to 890 MHz. Because the PIM array contains the BRAM, it cannot run faster than the BRAM Fmax. It passed the timing at 737 MHz, which is the target clock for Alveo U55 according to the proposed Ideal Standard. As observed in Table 16, the logic utilization of the controller is around 5% of the entire tile and requires no DSPs, while around 90% of the logic resources are consumed by the PIM array. Thus, the controller and the fanout tree are not expected to bottleneck system frequency or utilization. The GEMV tile's speed and scalability are fundamentally dependant on the PIM array, which is the desired outcome.

5.2.3 Scalability Study

To evaluate the scalability of our architecture on different device families, we followed the same approach as for PiCaSO. Along with Alveo U55, four representatives were selected from AMD's Virtex-7 and UltraScale+ devices based on two criteria: BRAM capacity and LUT-to-BRAM ratio. Table 17 lists these devices with their BRAM capacity, LUT-to-BRAM ratio, and a short ID used in Fig. 16. These are the same devices on which the scalability of PiCaSO-F was studied.

Table 17: Representatives of Virtex-7 and UltraScale+ Families [2]

Device	Tech	BRAM#	Ratio[1]	Max PE#[2]	ID
xcu55c-fsvh-2	US+	2016	646	64K	U55
xc7vx330tffg-2	V7	750	272	24K	V7-a
xc7vx485tffg-2	V7	1030	295	32K	V7-b
xc7v2000tfhg-2	V7	1292	946	41K	V7-c
xc7vx1140tflg-2	V7	1880	379	60K	V7-d
xcvu3p-ffvc-3	US+	720	547	23K	US-a
xcvu23p-vsva-3	US+	2112	488	67K	US-b
xcvu19p-fsvb-2	US+	2160	1892	69K	US-c
xcvu29p-figd-3	US+	2688	643	86K	US-d

[1] LUT-to-BRAM ratio

[2] Number of PEs utilizing all BRAMs as PIMs

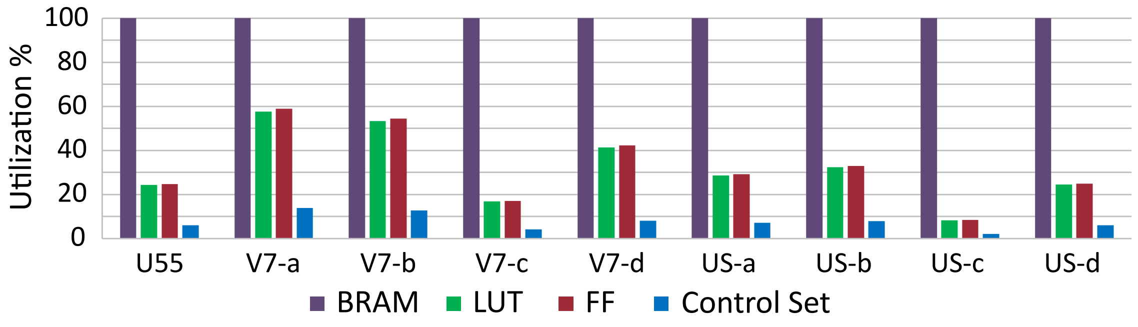


Figure 16: Resource usage of IMAGine on representatives of Virtex-7 and UltraScale+ families utilizing 100% BRAMs as PIM overlays.

The target clock frequency of the system was set to 100 MHz on all devices to avoid timing issues and only focus on the logic utilization of the system at this point.

Fig. 16 shows a bar graph of post-implementation utilization numbers of IMAGine on the representative devices. As observed, IMAGine can utilize 100% of the available BRAMs as PIM overlays providing 64K PEs in U55, with only 25% logic and 6% control set utilization. This leaves sufficient logic resources to implement the fanout trees and pipeline stages if they are needed to achieve the target clock speed. In fact, IMAGine scaled up to 100% of available BRAM in all the representative devices for Virtex-7 and UltraScale+ families.

In the Virtex-7 family, the device V7-a has the smallest number of BRAMs and the smallest LUT-to-BRAM ratio. IMAGine used around 60% logic resources to provide 24K PEs in V7-a. In the UltraScale+ family, US-a and US-b have the smallest number of BRAMs and the smallest

LUT-to-BRAM ratio, respectively. In these devices IMAGine provide 23K and 67K PEs, respectively, using roughly 30% logic resources. For devices with more BRAMs and a higher LUT-to-BRAM ratio the logic utilization is very small: the logic utilization in US-c is less than 10% providing 69K PEs.

If we can keep this scaling and run the PIM blocks at BRAM Fmax, the Ideal Standard of linear scaling of peak performance would be met. Compared to the scalability study presented in Section 4.2.3, the logic utilization of IMAGine is roughly 11% more than the standalone PiCaSO array on average across all the devices. This increase is due to the controller logic, data pipelines, and the additional logic implemented in PiCaSO-IM.

5.2.4 System-Level Timing Optimizations

For the final implementation, the target clock was set to the Ideal Standard for Alveo U55 with a period of 1.356 ns to match the BRAM Fmax. The goal of the study was to find out how close we can get to the Ideal Standard, and what are the practical challenges that limit us from achieving the Ideal Standard. Achieving the best performance on a device always requires several iterations of device-specific optimizations. In the first iteration, we started with GEMV tiles having a 4×4 PIM array, without the fanout tree between the controller and the array. After going through the implementation flow with the default settings of Vivado and a few optimization iterations, we achieved a setup slack of -0.52 ns. The critical paths were within the controller with a logic depth of 4, going through the pipeline stage A of the controller as shown in Fig. 15(a). So, we enabled the pipeline stage A in the controller and moved forward with the second iteration.

At the end of the second iteration of implementation, we achieved a setup slack of -0.38 ns. The critical nets were the control signals between the controller and the PIM array. These nets were failing the timing requirement because of high fanout and long routes of the control signals between the controller and the PIM array. Thus, we synthesize a fanout tree between the controller and the PIM array empirically choosing 2 levels and a fanout of 4 for the next iteration.

The design achieved a setup slack of -0.27 ns in the third iteration. This time, we had to take

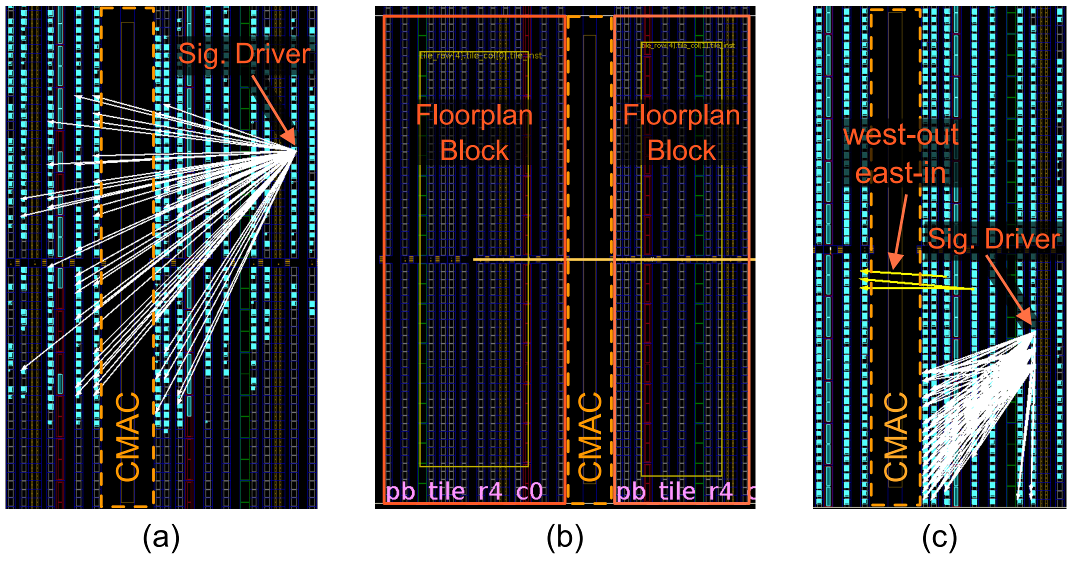


Figure 17: Avoiding unnecessary hard-block (CMAC) crossing (a) placement and net connections before floorplanning, (b) floorplanning to localize logic and routing, (c) placement and net connections in the final design.

a closer look at the design to reveal the main reason for the timing failures. Alveo U55 contains several hardened blocks like PLLs, high-performance IO blocks, PCIe blocks, high-performance ethernet port (CMAC) [102], etc. Most of the failing paths were due to the long routes crossing such hard blocks. The white lines in Fig. 17(a) highlight some of those critical nets crossing a CMAC block. To avoid placement results creating such paths, we created floorplanning blocks for each tile to localize the placement of logic and routing within a region dedicated to the tile. The floorplanning blocks were placed avoiding those hard blocks as shown in Fig. 17(b). This required defining a tile with the PIM array dimension of 12×2 for Alveo U55.

Fig. 17(c) shows the placement and net connections in the final iteration. The logic and routing of each tile were localized on either side of the hard block; the white lines representing the high fanout control signals do not cross the CMAC block. Only the logically essential nets, the inter-tile connections for east-to-west accumulation, cross the CMAC block requiring minimal routing resources. The yellow lines in Fig. 17(c) highlight some of those inter-tile nets.

All paths in the final design met the timing requirement at 737 MHz clock, which demonstrates that the ideal clocking standard is practically achievable. Utilizing 100% available

BRAMs as PIMs, this design also achieved the ideal linear scaling of peak-performance. Surprisingly, this clock rate is faster than custom GEMM accelerator ASICs TPU v1-v2 [103] and Alibaba Hanguang 800 [104], that run at 700 MHz. Both Alveo U55 and TPU v2 are manufactured at 16 nm and Hanguang 800 at 12nm technology nodes. So, this clock improvement of IMAGine is not due to a technology node difference. On Alveo U55, IMAGine has an equal number of PEs compared to TPU v1 (64K), and 4$\times$ of TPU v2 (16K). However, IMAGine can only deliver up to 0.33 TOPS which is significantly smaller compared to TPU v1 (92 TOPS) and v2 (46 TOPS) due its bit-serial architecture. This makes a compelling case: even though an FPGA design will probably never outperform custom ASICs in terms of peak-performance or performance-per-watt, the right set of design goals and guiding principles can bring it very close in terms of clock speed and compute density. Our proposed Ideal Standard can serve that purpose for PIM array-based FPGA designs.

5.3 Comparison With Other PIM Accelerators

Table 18 shows the utilization and system frequencies of existing GEMV engines and equivalent PIM-based systems. System-level utilizations and frequencies for BRAMAC and M4BRAM-based systems were not reported in [81, 82]. Though RIMA is specialized for accelerating RNNs, a major part of the system implements GEMV operation using Dot Product Engines (M-DPEs) [84]. The RIMA numbers are taken from Table II of [84] for comparison, which was evaluated on a Stratix 10 GX2800 FPGA with a BRAM Fmax of 1 GHz [87]. Its fastest reported configuration (RIMA-Fast) runs at 455 MHz, which is 2.2$\times$ slower than the BRAM Fmax. The largest reported configuration (RIMA-Large) utilizes 93% of BRAMs and runs at 278 MHz, 4$\times$ slower compared to BRAM Fmax. The GEMV/GEMM systems based on CCB and CoMeFa were evaluated on an Arria 10 GX900 with a BRAM Fmax of 730 MHz [1]. Though CoMeFa-based designs run slightly faster than the CCB-GEMV engine, they are still roughly 3$\times$ slower than the BRAM Fmax of the device. Thus, neither of the CCB and CoMeFa-based GEMV/GEMM engines scaled well at the system level.

Table 18: Utilization and Frequency of PIM-based GEMV/GEMM Engines

	LUT	FF	DSP	BRAM	f_{Sys}[1]	Rel. Freq
RIMA-Fast	60%		50%	55%	455	45.5%
RIMA-Large	89%		50%	93%	278	27.8%
CCB GEMV	27.9%		90.1%	91.8%	231	31.6%
CoMeFa-A GEMV	27.9%		90.1%	91.8%	242	33.2%
CoMeFa-D GEMM	25.5%		92.4%	86.7%	267	36.6%
SPAR-2 (US+)	11.3%	2.4%	0.0%	14.5%	200	27.1%
SPAR-2 (V7)	28.5%	7.0%	0.0%	30.4%	130	23.9%
IMAGine	35.6%	24.8%	0.0%	100.0%	737	100.0%
IMAGine-CB[2]	10.1%	7.2%	0.0%	100.0%	737	100.0%

[1] System frequency in MHz
[2] IMAGine with custom-BRAM PiCaSO-F (PiCaSO-CB)

SPAR-2 [29] utilized only 30% of the BRAMs while running 4× slower than BRAM Fmax on both tested platforms. Thus, its performance and scalability are even worse than CCB and CoMeFa-based systems. On the other hand, IMAGine has a system clock running at the BRAM Fmax while utilizing 100% device BRAM as PIMs. Thus IMAGine takes advantage of the full internal bandwidth offered by the BRAMs in the device. As a PIM-based GEMV engine, IMAGine not only outperformed all existing designs but also verified that the Ideal Standards of clocking and scalability are achievable. This is an important proof of concept design that dispels earlier beliefs that overlays cannot achieve BRAM Fmax clock frequencies at the system level. It is the fastest PIM-based GEMV engine implemented on any FPGA, running at a clock rate 2.65× – 3.2× faster than any existing design.

As observed in Table 18, RIMA and CCB/CoMeFa-based GEMV engines exhaust either the logic resources or the DSPs of the device even though their PIM blocks are implemented by customizing the BRAM tile itself. Even after being an overlay, IMAGine is achieving faster clock and better scalability using 0 DSPs and only one-third of the device logic resources due to its near-optimal architectural choices guided by the proposed Ideal Standard. Like SPAR-2, IMAGine does not use DSPs to implement the bit-serial PEs. With a custom-BRAM PIM module like PiCaSO-CB discussed in Section 5.1.4, IMAGine would consume about 10% of device resources while being fully scalable and implementable even in resource-limited FPGAs.

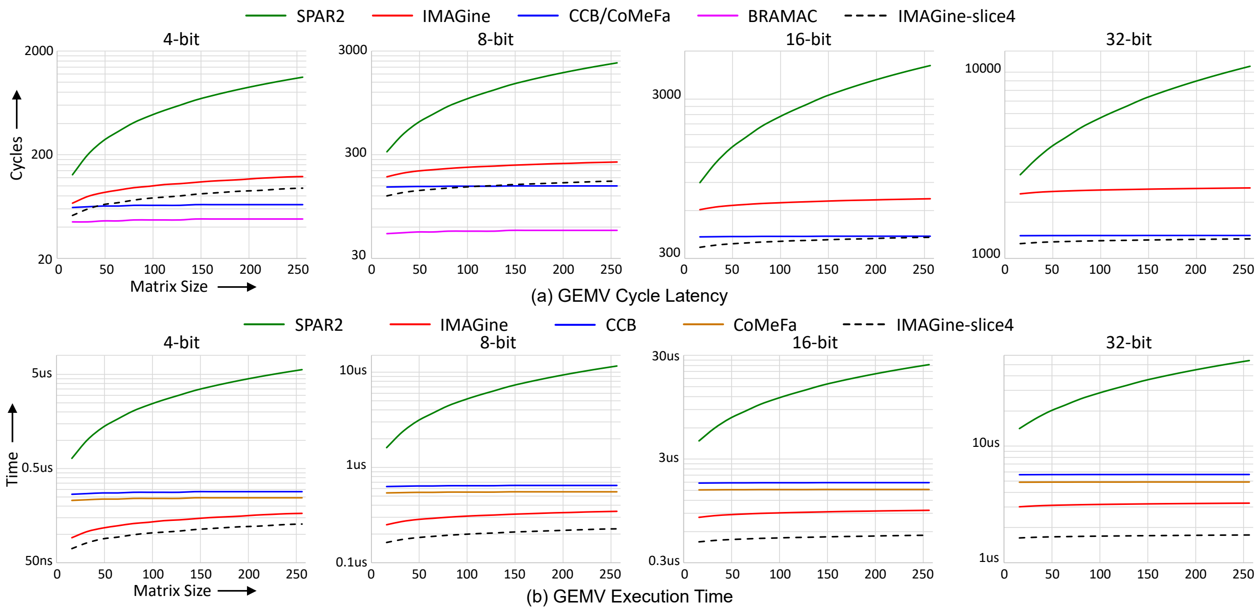


Figure 18: Cycle latency and execution time of GEMV operation on different PIM array-based FPGA accelerators

All implementations of SPAR-2 and IMAGine use 0% of the DSPs, which means DSPs are not essentially required in PIM accelerators. Thus, some or most of the DSPs can be replaced with BRAMs/PIM blocks to further improve the peak performance of the PIM accelerators in FPGA families targeting PIM designs.

5.4 GEMV Execution Latency

Fig. 18(a) displays GEMV cycle latency for the PIM designs showing matrix dimensions (matrices are square) on the x-axis and cycle latency in log scale on the y-axis. The execution times shown in Fig. 18(b) are computed by multiplying cycle latencies with the corresponding clock periods of CCB GEMV, CoMeFa-D GEMM, SPAR-2 (US+), and IMAGine from Table 18 system frequencies. We adopted the approach in [81] to model the block-level cycle latencies of CCB, CoMeFa, BRAMAC, and SPAR-2 using their analytical models. IMAGine's latency model was developed and validated through cycle-accurate simulations of the system. The global reduction tree of RIMA was modeled as an adder tree, perfectly pipelined with the last iteration of in-block accumulation. For CoMeFa and BRAMAC GEMV latency, we assumed the same system as RIMA but with CoMeFa or BRAMAC replacing the CCB blocks. For SPAR-2, only the latency for the binary-add version is shown because the linear-add version is too slow to plot together with the other systems.

As observed in Fig. 18(a), BRAMAC has the shortest cycle latency, due to their hybrid bit-serial & bit-parallel MAC2 algorithm. MAC latency in BRAMAC grows linearly with operand bit-width, while it grows quadratically in bit-serial architectures like CCB, CoMeFa, SPAR-2, and PiCaSO. This result is not surprising; domain-specific architectures can deliver the best performance at a very low cost. BRAMAC supports only 2, 4, and 8-bit precisions particularly targetting low-precision applications like quantized neural network acceleration. However, BRAMAC is less suitable for general computing tasks like GEMV for full-precision scientific computing or even neural networks requiring wider precisions. BRAMAC [81] did not report their system-level frequency which is why we could not plot its execution time.

In all precisions, SPAR-2 has the longest cycle latency and execution time due to its slow NEWS accumulation network. Its accumulation latency increases almost linearly with the matrix dimension. CCB and CoMeFa-based GEMV engines have the shortest cycle latency among bit-serial architectures across all precisions. This is due to their fast reduction algorithm based on a pop-count adder and pipelined adder tree. The cycle latency of IMAGine is significantly shorter compared to SPAR-2 but longer than CCB/CoMeFa-based implementations. However, IMAGine has the fastest clock, which is at a minimum $2\times$ faster than any of the other GEMV engines. When accounting for the clock frequency, IMAGine outperforms all other GEMV engines in terms of overall execution time. This highlights the importance of the system clock speed over the cycle latency. Although the reduction tree-based approach in the CCB/CoMeFa GEMV engines has the shortest cycle latency, the slower clock ultimately degrades the end-to-end latency times below that of IMAGine.

Table 19: Curve-Fitted Parameters of Eqn. (1) for 32-bit Accumulation

	Fitted Value			Speed Interpretation	
	a	b	c	Addition (a)	Movement (b)
SPAR-2 Linear-Add	0	96	0	Very Slow	Very Slow
SPAR-2 Binary-Add	2	32	0	Standard	Very Slow
CCB/CoMeFa	0.03	0.02	203.1	Fast	Fast
IMAGine	1.2	0.9	143	Standard	Standard

5.5 Further Improvements

While IMAGine outperforms the existing designs, this does not guarantee it is the optimal implementation. There could still be some room for improvement, particularly in the reduction network. When examining all of the architectures, we can intuitively understand which designs are better suited for the reduction process. However, the proposed Ideal reduction-latency Standard discussed in Chapter 3 can quantify these insights even without the architectural knowledge of the implementations. Since the studied systems employ different reduction approaches, we define the reduction latency for the GEMV operation as any cycle spent outside the multiplication stage, encompassing the In-Block latency (2). We curve-fitted the proposed reduction-latency model (1) to the reduction cycle latencies of the designs in Fig. 18(a). Table 19 shows the fitted parameters for 32-bit operands and their interpretations.

SPAR-2 linear-add parameters significantly deviate from their ideal ranges outlined in Table 4, revealing a suboptimal reduction network design. SPAR-2 binary-add has a in the standard range, which means the reduction operation is at least near optimal. This was achieved through an optimization reported in [3] to reduce the number of add operations. A high value of b in both approaches indicates that accumulation latency is dominated by the data movement part, which is notably slower than the ideal. The value of c being 0 indicates the reduction process does not involve extra cycles outside the reduction network, which is true by design in SPAR-2.

For CCB/CoMeFa, both parameters are near the smallest possible values: here $0.3 \approxeq 1/N$, N=32. This implies it has the shortest possible cycle latency. The value of c approximates the In-Block accumulation and pipeline setup latency of 202 cycles spent generating the 16 partial sums per block [84] before they enter the reduction network.

IMAGine's parameters fall within the standard range, implying at least a near-optimal implementation. In this case, as well, the value of c approximates the In-Block accumulation latency of 144 cycles. Since b is near its upper bound, it indicates room for improvement in the data movement part. Because IMAGine is utilizing only 30% of the logic resources in U55, we can modify the network to implement 2-bit or 4-bit bit-sliced accumulation, possibly without affecting the system clock speed. This has the potential to further improve the cycle latency of the GEMV operation. Additionally, the PEs can be modified to implement Booth's Radix-4 instead of the default Radix-2 algorithm adopted from PiCaSO.

The IMAGine-slice4 curves in Fig. 18 shows the GEMV latency of a variant of IMAGine with a 4-bit sliced accumulation network and a PE implementing Booth's radix-4 multiplication. This latency is estimated by adjusting the analytical model of IMAGine assuming no effect on the clock rate. In terms of cycle latency, it can run almost as fast as CCB/CoMeFa-based GEMV implementations. Because of the higher system frequency, it will then significantly outperform all other state-of-the-art PIM-based GEMV accelerators. This example demonstrates how the proposed Ideal Standard can aid in identifying design inefficiencies and making optimal design choices for PIM-based accelerators in FPGAs.

5.6 Discussion

The implementation and analysis of IMAGine demonstrated that the full internal BRAM bandwidth of FPGAs can be exploited by operating at the maximum BRAM frequency in a practical PIM array accelerator. Scalability studies showed linear scaling of processing capacity with increasing BRAM density, even for devices with low LUT-to-BRAM ratios. IMAGine, a PIM array-based GEMV accelerator with 64K PEs running on Alveo U55, achieved clock speeds faster than the Tensor Processing Unit (TPU v1-v2) and Alibaba Hanguang 800. This challenges the long-held belief that FPGA overlays and reconfigurable fabrics must clock slower than ASIC designs. Although IMAGine outperformed state-of-the-art FPGA-based PIM architectures and clocked faster than some ASIC accelerators, further improvements are necessary. Enhancing end-to-end GEMV latency could be achieved by overlapping data broadcasting with multiply-accumulate operations. Additionally, IMAGine requires further development to support point-wise operations and non-linear activation functions, which are crucial for deep-learning applications.

Chapter 6

DA-VinCi: A Deeplearning Accelerator Overlay using in-Memory Computing

In the last chapter, we demonstrated how to use PiCaSO to build an Array Processor, IMAGine, to accelerate GEMV computation. While IMAGine effectively accelerates the GEMV/GEMM components of deep-learning models, the operations on the resultant vector of GEMV/GEMM still require CPU processing. Significant parallelism exists within these vector operations, presenting an opportunity to further accelerate deep learning models. However, the sequential nature of a CPU prevents it from exploiting this parallelism effectively. Additionally, non-linear activation functions such as Sigmoid and Tanh can take 10 to 100 cycles on a CPU, whereas these functions can be computed in just a few cycles using a custom ALU. In this chapter, we will extend IMAGine from a GEMV accelerator to a comprehensive deep-learning accelerator by incorporating additional features that fully leverage the parallelism in vector operations. We name this DA-VinCi, a Deeplearning Accelerator Overlay using in-Memory Computing.

6.1 Architecture

6.1.1 System-Level Architecture

Figure 19 illustrates the system-level architecture of DA-VinCi, which closely resembles IMAGine, with key differences including the vector-vector engine (VV-Engine) replacing the column-shift-registers and an enhanced front-end interface. These new components are highlighted in the figure using colors, while the common elements with IMAGine are grayed out for clarity. The front-end interface receives DA-VinCi instructions via an input FIFO, decodes them, and then redirects them to either the GEMV Engine or the VV-Engine. Post GEMV operation, the resultant vectors are transferred to the VV-Engine, where vector-vector point-wise operations and activation functions are performed on them. Finally, the processed data is shifted out to the output FIFO through a chain of shift-registers in the VV-Engine.

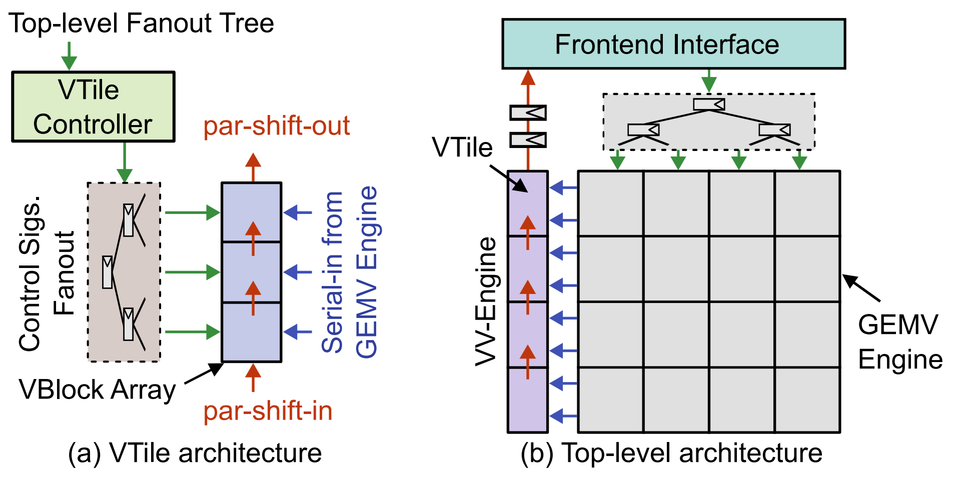


Figure 19: System architecture of DA-VinCi illustrating the data and instruction flow (a) VV-Engine Tile (VTile) architecture (b) System-level architecture.

6.1.2 Vector-Vector Engine Architecture

Figure 19(a) illustrates the architecture of a VV-Engine tile (VTile), which is designed to process one-dimensional vectors and bears a close resemblance to a GEMV tile. Each VTile features a tile controller that connects to an array of vector-vector processing blocks (VBlocks) via a fanout tree of control signals. Unlike the PIM-Array in GEMV tiles, the VBlock array in a VTile is one-dimensional, reflecting its specialized function for handling vector-vector operations. This streamlined design allows the VTile to perform efficient point-wise operations and activation functions on the resulting vectors from the GEMV Engine exploiting their parallelism.

Figure 20(a) shows the architecture of a VBlock, the main computing component of the VV-Engine. Each VBlock comprises several modules: a BRAM, an ALU with an output register (OREG), a serial-parallel shift register (SREG), and a lookup mechanism to support non-linear activation functions. The BRAM plays a dual role within the VBlock, with its lower half functioning as a register file and its upper half serves as lookup tables for several non-linear activation functions. After the GEMV Engine processes the data, the resultant vector is copied to the SREG. This vector can then be used as an operand for the ALU or transferred to the registerfile for future operations. SREGs of adjacent VBlocks in a tile are connected to form a parallel shifting chain (SREG-chain), by connecting their par-shift-in and par-shift-out signals.

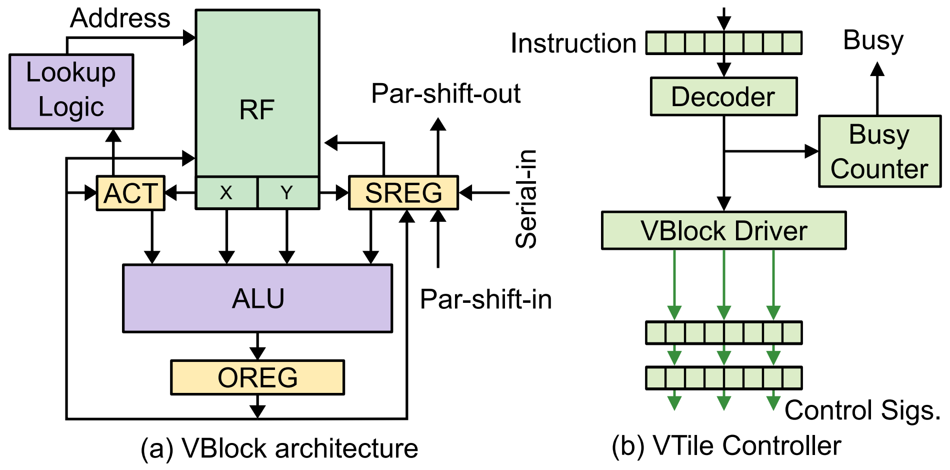
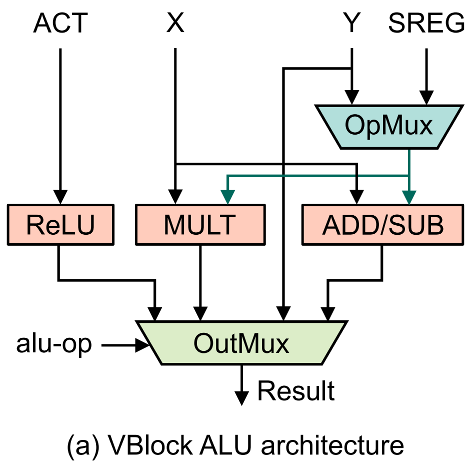


Figure 20: The architecture of VTile components (a) VBlock architecture (b) VTile Controller

This chain runs through the entire VV-Engine, which is used to shift out the result vector to the FIFO-out through the front-end interface.

Figure 20(b) shows the architecture of the VTile controller, which is a streamlined version of the GEMV tile controller. The VTile controller's simplified design includes a VBlock driver module that performs the same function as the single-cycle driver in the GEMV tile controller. Unlike the GEMV controller, the VTile controller does not include a multi-cycle FSM driver. Instead, multi-cycle instructions are managed by asserting the control signals for the VBlock array for the required number of cycles. This task is accomplished by the Busy-Counter module, which ensures that the necessary control signals are maintained over the appropriate number of cycles before fetching the next instruction, to complete the multi-cycle instructions effectively. This architecture allows for efficient handling of vector operations within the VTile, maintaining high performance while simplifying the control logic.

Figure 21(a) shows the architecture of the VBlock ALU. The ALU is composed of an adder/subtractor module, a fixed-point multiplier implemented using a DSP block, and a combinatorial block that executes the ReLU activation function. The ALU is designed to take four inputs: the activation function input (ACT), port X and port Y from the BRAM, and the shift-register (SREG). Depending on the ALU opcode, a subset of these inputs is selected to perform the desired computation. All of the computed results go to an output multiplexer



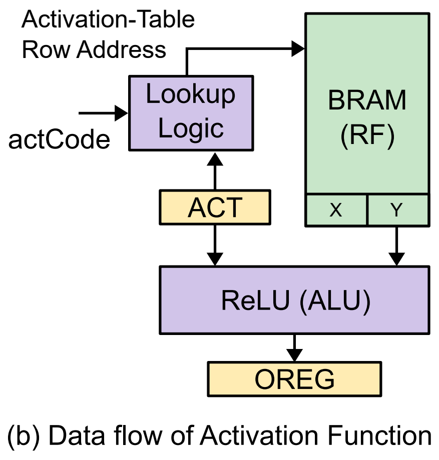

Activation-Table Row Address
Lookup Logic
actCode
BRAM (RF)
X
Y
ACT
ReLU (ALU)
OREG
(b) Data flow of Activation Function

Figure 21: VBlock computing architecture (a) ALU architecture (b) Dataflow of activation functions

(OutMux) before leaving the ALU. Based on the selected ALU-opcode, one of these results is selected as the ALU output result. Additionally, the Y input is directly connected to the OutMux, providing a direct data path that facilitates copying data from one register to another within the registerfile through OREG. Additional simple activation functions like ReLU can be supported by adding more logic blocks and connecting the ACT input to it. This modular design ensures efficient and flexible data handling, allowing the ALU to perform a variety of arithmetic and activation functions critical for deep-learning applications.

Figure 21(b) shows the data flow of the non-linear activation function computation. Simple activation functions, like ReLU, are performed in the ALU taking the ACT register as the input. The non-linear activation functions are computed using a lookup-based approach, which makes this architecture very flexible compared to the fixed-function architectures of activation functions [3, 25, 29]. During the initialization of DA-VinCi, the BRAM is preloaded with lookup tables of all the non-linear activation functions needed for the target applications. To compute the activation output of an input, the input value is first copied to the ACT register. Based on the value in the ACT register and an activation code (actCode) from the instruction word, a row address is generated for the BRAM. This row address points to the non-linear activation output of the value in the ACT register for the activation function corresponding to the actCode. The output is then stored in the OREG. This entire lookup process is completed in three cycles.

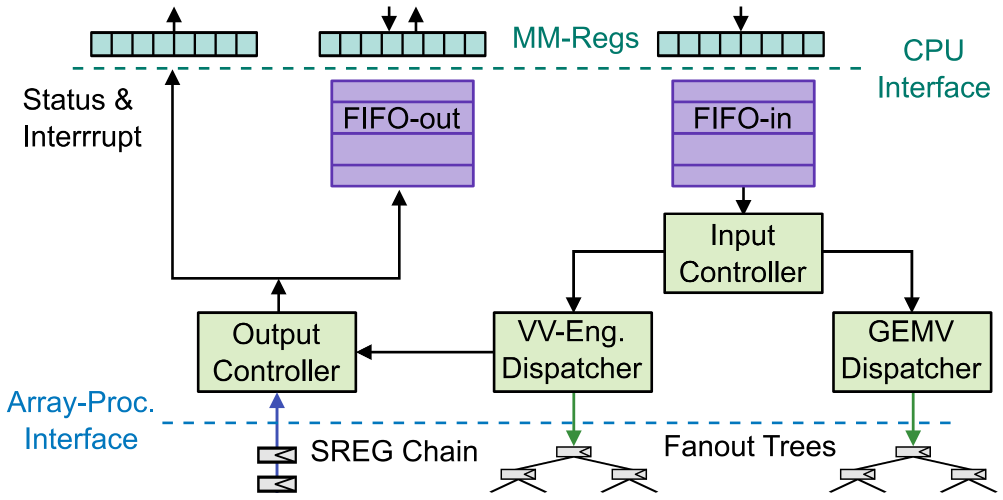


Figure 22: DA-VinCi front-end interface architecture

6.1.3 Front-end Interface Architecture

Figure 22 illustrates the architecture of the front-end interface of DA-VinCi. This interface comprises several key components: memory-mapped registers, two FIFOs, status registers, and various controller blocks. The memory-mapped registers are directly accessible to the CPU, allowing it to write data and control words to the interface FIFOs and read outputs and status of the accelerator. The FIFO-in is responsible for receiving 32-bit instruction words from the memory-mapped register, while the FIFO-out holds the most recent resultant vectors that have been shifted out from the VV-Engine. The last element of the vector written to the FIFO-out generates an interrupt signal for the CPU, indicating that the result vector is ready for the CPU to read out.

The input controller is responsible for fetching instructions from the FIFO-in, examining the opcode to determine whether the instruction is meant for the GEMV Engine or the VV-Engine, and then initiating a handshake with the appropriate dispatcher module. The GEMV and VV-Engine dispatchers manage the states of their respective compute engines, handling handshake requests from the input controller. They only accept a new instruction if the corresponding engine is idle or will be idle in the next cycle, ensuring the proper synchronization of the operations.

Table 20: Instruction Set of DA-VinCi: GEMV Engine Subset

Instruction	Operands	Semantics
mv_write	addr, data	BRAM[addr] <= data
mv_nop	-	Spend 1 cycle doing nothing
mv_mov	rd, rs	RF[rd] <= RF[rs]
mv_add	rd, rs1, rs2	RF[rd] <= RF[rs1] + RF[rs2]
mv_sub	rd, rs1, rs2	RF[rd] <= RF[rs1] - RF[rs2]
mv_selectBlk	rowID, colID	select BRAM at GEMV[rowID, colID] for mv_write
mv_selectRow	rowID	select all BRAMs in the GEMV[rowID] for mv_write
mv_selectCol	colID	select all BRAMs in the GEMV[:, colID] for mv_write
mv_selectAll	-	select all BRAMs in the GEMV array for mv_write
mv_accumRow	level, reg	row-wise accumulation of RF[reg] at L=level
mv_updatepp	ppreg, mpcand, mplier, mbit	RF[ppreg] += RF[mpcand]$\times$RF[mplier][mbit]

[*] BRAM[addr] semantics access BRAMs directly using row address
[*] RF[reg] semantics access BRAMs as PE registerfiles, each register spanning multiple rows

The output controller monitors the shifted-out vectors through the SREG chain of VV-Engine and sets the interrupt flag when the last element is shifted out, signaling the completion of the operation. Additionally, it manages various status and error flags communicating with other modules, including those for invalid instructions, FIFO-out overflow, and lost instructions, to ensure the reliable and accurate operation of the system.

6.1.4 DA-VinCi Instruction Set

DA-VinCi has its own instruction set architecture (ISA). This ISA is defined independently of the hardware implementation of submodules such as the GEMV and VV-Engines using the front-end interface module. The front-end interface handles the 32-bit instruction word and translates it into the corresponding instruction format for the respective module. This abstraction layer ensures that the instruction set and compiled programs are implementation-independent and portable across future generations of DA-VinCi.

Table 20 lists the subset of the ISA that controls the GEMV engine. These assembly

Table 21: Instruction Set of DA-VinCi: VV-Engine Subset

Instruction	Operands	Semantics
vv_write	addr, data	BRAM[addr] <= data
vv_nop	-	Spend 1 cycle doing nothing
vv_mov	rd, rs	RF[rd] <= RF[rs]
vv_add	rs1, rs2	OREG <= RF[rs1] + RF[rs2]
vv_sub	rs1, rs2	OREG <= RF[rs1] - RF[rs2]
vv_mult	rs1, rs2	OREG <= RF[rs1] $\times$ RF[rs2] [a]
vv_activ	actCode	OREG <= activation-func<actCode>(ACT) [b]
vv_shiftOff	-	Disable all shifting of SREG
vv_serialEn	-	Enable serial shifting of SREG
vv_parallelEn	-	Enable parallel shifting of SREG chain
vv_selectBlk	blkID	Select BRAM at VV-Engine[blkID] for vv_write
vv_selectAll	-	Select all BRAMs in the VV-Engine for vv_write

[*] BRAM[addr] semantics access BRAMs directly using row address
[*] RF[reg] semantics access BRAMs as vector registerfiles
[a] Fixed-point multiplication
[b] activation-func<actCode> selects between ReLU or non-linear lookup tables

instructions are prefixed with "mv_" to denote their function in performing matrix-vector operations. The ISA includes instructions for selectively writing to BRAMs in the GEMV array, moving data within the registerfile, performing arithmetic operations like ADD and SUB, and executing row-wise accumulation.

One particularly noteworthy instruction is the update-partial-product (updatepp) instruction. This instruction executes one step of Booth's multiplication algorithm and updates the partial-product register (ppreg). It achieves this by adding the partial-product register (ppreg) with the product of the multiplicand register (mpcand) and the multiplier bit (mbit). It performs the necessary shifts of the operands required by Booth's algorithm. Multiple iterations of this instruction, using successive values of mbit, are utilized to perform the complete multiplication between the mpcand and mplier registers in the GEMV Engine.

Table 21 lists the subset of the ISA that controls the VV-Engine. These assembly instructions are prefixed with “vv_” to distinguish them from those in the GEMV subset. Similar to the GEMV subset, the VV-Engine ISA includes instructions for selectively writing to BRAMs in the VBlock array, moving data between registers, and performing essential computations such as ADD, SUB, and MULT, as well as activations. It is important to note that the multiplication performed by the ALU is a fixed-point multiplication. The “vv_active” instruction is particularly noteworthy, as it allows for the selection of either one of the non-linear activation functions stored in the BRAM or the ReLU function using the appropriate ALU opcode. If one or more simple functions like ReLU are added to the ALU, only some extra actCode will need to be added to the assembler to support them. This along with reloadable non-linear lookup tables provides the flexibility needed for a wide range of activation functions, which is critical for deep-learning applications.

Unlike the GEMV subset, the VV-Engine instructions also include instructions to control the shift-register chain within the VV-Engine. Serial shifting can be enabled to copy data from the GEMV Engine, while all shifting can be disabled to retain the content during computation. Parallel shifting is enabled to push out the result vector to the FIFO-out. These instructions provide complete control over the data flow within the VV-Engine essential for maintaining high efficiency and performance.

These comprehensive instructions collectively make the VV-Engine highly flexible, and capable of adapting to various computational needs. This combination of flexibility, performance, and scalability positions the VV-Engine as a powerful component within the DA-VinCi architecture, capable of meeting the demanding requirements of current and future deep-learning applications.

6.2 Implementation and Analysis

6.2.1 VV-Engine Tile

Before implementing the VV-Engine, a single VV-Engine tile (VTile) was studied to ensure it meets the requirements for the Ideal Design Standards. A tile dimension of 12×1 was chosen for implementation on the Alveo U55 to match the tile height of the GEMV tiles, simplifying the

Table 22: Utilization of 12×1 VV-Engine Tile Components on Alveo U55 at 737 MHz

	Controller	Tile Rel.	Fanout	Tile Rel.	VBlock-Array	Tile Rel.	Tile
LUT	27	1.7%	0	0.0%	1580	98.3%	1607
FF	116	8.5%	247	18.1%	1004	73.4%	1367
DSP	0	0.0%	0	0.0%	12	100.0%	12
BRAM	0	0.0%	0	0.0%	12.0	100.0%	12

connection between the VV-Engine and the GEMV-Engine. The VTiles also have two stages of pipelines in the fanout tree between the controller and VBlock array, which match the instruction propagation latency of the GEMV tiles. This design choice simplifies the synchronization between the VV-Engine and the GEMV-Engine.

Table 22 provides a detailed breakdown of the resource utilization for a 12×1 VTile and its individual components when implemented on the Alveo U55 at 737 MHz, the BRAM Fmax. Comparing these utilization figures with the utilization of GEMV tile in Table 16 offers valuable insights into the efficiency and resource allocation of the VV-Engine design. One notable change in the VTile design is its significantly smaller controller size compared to that of the GEMV tile controller. This reduction is attributed to a simpler design without multi-cycle FSMs. Additionally, the fanout tree within the VTile exhibits lower flip-flop utilization, due to the fewer controller signals compared to GEMV tiles.

The VBlock array, comprising 12 VBlocks, employs 12 DSPs dedicated to handling fixed-point multiplications. Each VBlock utilizes a RAMB36 tile as its register file, resulting in a BRAM utilization of 12 at the tile level. The 12×1 VTile design successfully meets timing constraints with a positive slack at BRAM Fmax. This ensures that the VV-Engine adheres to ideal clocking goals maintaining high performance and scalability standards.

6.2.2 Front-End Interface

Table 23 shows the utilization of the front-end interface implemented on the Alveo U55 at BRAM Fmax. The “Available” column indicates the total number of each resource available in the U55, while the “Utilization%” column reflects the device-level utilization of the front-end interface. As

Table 23: Utilization of the front-end interface on Alveo U55 at 737 MHz

	Utilization	Utilization%	Available
LUT	50	0.004%	1303680
FF	32	0.001%	2607360
DSP	0	0.000%	9024
BRAM	1	0.050%	2016

discussed in Section 6.1.3, the design of the front-end interface is very simple, resulting in minimal usage of LUTs and flip-flops. Since the interface handles straightforward instruction management tasks without complex computations, its DSP utilization is 0. FIFO-in and FIFO-out, used as the interface with the CPU, are implemented using 2 RAMB18, consuming 1 RAMB36 tile. Such low utilization makes it a lightweight abstraction layer, ensuring portability and binary-code compatibility across future DA-VinCi implementations at almost zero cost.

The front-end interface successfully meets timing requirements at BRAM Fmax. The negligible device-level utilization along with its compliance with the ideal clocking goal ensures that DA-VinCi can achieve its high-performance and scalability goals at the system level.

6.2.3 System-Level Analysis

With the submodules of DA-VinCi meeting Ideal Design Standards, we move forward with the system-level implementation on the Alveo U55 platform. The dimensions of the GEMV engine and VV-Engine were chosen to utilize the 100% available BRAM resources, operating at the target clock speed of 737 MHz, the BRAM Fmax. Similar to IMAGine, the process involved multiple iterative implementations to achieve these design goals.

Placement and routing optimizations, as detailed in Section 5.2.4, were employed to achieve target performance and scalability. This included selecting GEMV tiles of 12$\times$2 dimensions and VTiles of 12$\times$1 dimensions on Alveo U55. Floorplan blocks were created to avoid hardened blocks like CMAC. Each tile's control and data signals were localized, allowing only the inter-tile network connections to cross the floorplan blocks. All paths in the final design met the timing requirement at 737 MHz clock, demonstrating DA-VinCi's adherence to the ideal clocking

Table 24: Utilization breakdown of the largest DA-VinCi (60K PEs) and its major components on Alveo U55 at 737 MHz (BRAM Fmax)

	GEMV-Engine	DA-VinCi%	VV-Engine	DA-VinCi%	DA-VinCi	Device%
LUT	463286	95.9%	19703	4.1%	482989	37.0%
FF	608812	97.4%	16202	2.6%	625014	24.0%
DSP	0	0.0%	144	100.0%	144	1.6%
BRAM	1872	92.9%	144	7.1%	2016	100.0%

standard. Furthermore, by utilizing 100% of available BRAMs, DA-VinCi achieved ideal linear scaling of peak performance.

Table 24 provides a detailed utilization breakdown of DA-VinCi and its major components. By utilizing 100% of the available BRAM, DA-VinCi offers 60K bit-serial PEs capable of performing the MAC operations involved in the GEMV/GEMM steps in deep learning applications. The BRAMs that provided the additional 4K PEs in IMAGine have been reallocated to the VV-Engine. A comparison of the DA-VinCi column in Table 24 with the IMAGine row in Table 18 reveals that system-level utilizations are almost identical, with the primary difference being the 1.6% DSP utilization for fixed-point multiplications in the VV-Engine.

The GEMV-Engine column indicates that over 90% of DA-VinCi's resources are dedicated to it, which is desired given that most of the parallelism in deep-learning applications arises from GEMV/GEMM computations. In contrast, the VV-Engine column shows that less than 10% of DA-VinCi's resources are allocated to it. Since the VV-Engine is the sole component utilizing DSPs, it accounts for 100% of the system's DSP utilization.

6.3 Comparison with the Benchmark Design

To study the performance gain of DA-VinCi on real-world applications, it was compared with three classes of accelerators: the benchmark overlay, PIM-based deep-learning accelerators, and custom FPGA accelerators. The comparison aims to highlight DA-VinCi's advantages in terms of latency. To ensure a fair and accurate comparison, utilization and latency were collected by running the same applications on similar, if not identical, FPGA platforms. This approach allows

Table 25: Application latency comparison of DA-VinCi with the benchmark overlay SPAR-2 [3]

Design	Latency[a]	Speedup	Precision[b]	LUT	FF	BRAM	DSP	Freq[c]	FPGA
LSTM-1 (61, 250, 250, 250, 39) on TIMIT dataset									
SPAR-2	11400	1	FxP32	133890	56207	313	0	200	US+
DA-VinCi	56.63	201.3	FxP32	521991	675180	2250	250	737	US+
LSTM-2 (64, 128, 128, 64) on CharRec dataset									
SPAR-2	257.1	1	FxP16	133890	56207	313	0	200	US+
DA-VinCi	18.72	13.7	FxP16	141095	178127	640	128	737	US+
MLP-1 (784, 100, 100, 10) on MNIST dataset									
SPAR-2	300	1	FxP32	133890	56207	313	0	200	US+
DA-VinCi	22.02	13.6	FxP32	98919	124198	450	100	737	US+
GRU-1 (39, 256, 200, 10) on DeepSpeech datase									
SPAR-2	2100	1	FxP16	133890	56207	313	0	200	US+
DA-VinCi	17.59	119.4	FxP16	534827	692309	2304	256	737	US+

[a] Execution latency in micro-seconds (us)
[b] Precision FxP := Fixed-Point
[c] System frequency in MHz

us to directly assess how DA-VinCi stacks up against other accelerators under comparable conditions, providing clear insights into its potential benefits and improvements in practical deep-learning applications.

Table 25 compares the latency and utilization of DA-VinCi with the benchmark overlay, SPAR-2, on Virtex UltraScale+ FPGAs. The LSTM-1 benchmark, used for speech recognition on the TIMIT dataset [105], has an input size of 61, three hidden layers of size 250, and a fully-connected output layer of size 39. The LSTM-2 benchmark, used for character recognition on the Shakespeare dataset [106], features an input size of 64, two hidden layers of size 128, and a fully-connected output layer of size 64. The MLP-1 benchmark, designed for handwritten digit recognition on the MNIST dataset [107], includes 784 inputs, two fully-connected layers of size 100, and an output layer of size 10. Lastly, the GRU-1 benchmark, used for speech recognition on the DeepSpeech dataset [108], consists of an input size of 39, two hidden layers of sizes 256 and 200, and a fully-connected output layer of size 10. These benchmarks are selected from the SPAR-2 latency study published in [3].

As observed from Table 25, DA-VinCi achieved a 201.3× speed-up on the LSTM-1 benchmark, reducing application latency from 11.4 ms to 56.63 us. This significant speed-up was achieved at the cost of higher utilization of system resources. On the LSTM-2 benchmark, DA-VinCi achieved a 13.7× speed-up, reducing latency from 257.1 us to 18.72 us, with similar logic utilization as SPAR-2. A similar speed-up was observed on the MLP-1 benchmark at a lower logic utilization. In LSTM-2 and MLP-1, the speed-up is attributed to DA-VinCi's more efficient computing architecture and superior system frequency compared to SPAR-2. Additionally, DA-VinCi achieved a 119.4× speed-up on the GRU-1 benchmark, with higher utilization similar to that in LSTM-1. In all cases, DA-VinCi outperformed SPAR-2 in system clock speed by 3.7×. This faster clock speed, combined with a scalable architecture, is the key to DA-VinCi's significant latency improvements over SPAR-2.

6.4 Comparison with PIM Accelerators

Table 27 compares the latency and utilization of DA-VinCi with PIM accelerators based on CCB and CoMeFa. RIMA, built on CCB, was evaluated on Stratix 10, while the CoMeFa-based GEMM engine was evaluated on Arria 10. These Intel device families are comparable to AMD's UltraScale+ family. The benchmarks were selected from the DeepBench [109] suite, as used in the original publications on RIMA [84] and CoMeFa [1]. The GEMM kernel (m=1536, k=512, n=32) involves multiplying a weight matrix of size 1536×512 with an input matrix of size 512×32, as described in the original publication [1]. The LSTM-3 (h=256) benchmark consists of an LSTM layer with 256 hidden units, the LSTM-4 (h=512) benchmark has an LSTM layer with 512 hidden units, and the GRU-2 (h=1024) benchmark includes a GRU layer with 1024 hidden units. The LSTM and GRU kernels were evaluated following the recommendations on the DeepBench webpage [109], excluding any input vector or output fully-connected layers.

As observed from Table 27, DA-VinCi achieved a 1.4× speed-up on the GEMM-1 benchmark compared to the CoMeFa-based GEMM engine. This improvement is due to DA-VinCi's faster system clock frequency and greater parallelism at the cost of higher resource

Table 26: Application latency comparison of DA-VinCi with PIM accelerators

Design	Latency[a]	Speedup	Preci.[b]	Logic[d]	FF	BRAM	DSP	Freq[c]	FPGA
GEMM-1 (m=1536, k=512, n=32) from Deepbench									
CoMeFa [1]	388.99[e]	1	INT8	86604	346413	2239	1317	267	Arria 10
DA-VinCi	268.68	1.4	INT8	534827	692309	2304	256	737	US+
LSTM-3 (h=256) from Deepbench									
RIMA [84]	60	1	INT8	559872	2239488	6447	2880	455	Stratix 10
DA-VinCi	2.52	23.8	INT8	534827	692309	2304	256	737	US+
LSTM-4 (h=512) from Deepbench									
RIMA [84]	20	1	INT8	690509	2762036	8088	2880	417	Stratix 10
DA-VinCi	9.27	2.2	INT8	534827	692309	2304	256	737	US+
GRU-2 (h=1024) from Deepbench									
RIMA [84]	2630	1	INT8	653184	2612736	7619	2880	417	Stratix 10
DA-VinCi	30.16	87.2	INT8	534827	692309	2304	256	737	US+

[a] Execution latency in micro-seconds (us)
[b] Precision INT := Integer
[c] System frequency in MHz
[d] ALMs for Stratix 10 and Arria 10, LUTs for UltraScale+
[e] Estimated using relative speed-up w.r.t CCB (RIMA) reported in [1]

utilization compared to the CoMeFa GEMM engine. On the LSTM-3 and LSTM-4 benchmarks, DA-VinCi demonstrated a 23.8× and 2.2× speed-up over RIMA, respectively. In these cases, DA-VinCi maintained similar logic utilization but significantly reduced the number of flip-flops, BRAM, and DSPs used.

Additionally, DA-VinCi achieved an impressive 87.2× speed-up on the GRU-2 benchmark, which is the largest of the three recurrent network models compared. This substantial speed-up illustrates DA-VinCi's ability to scale more effectively with increasing application size compared to RIMA. Notably, all RIMA implementations run at less than half the speed of the Stratix 10 BRAM with 1 GHz Fmax, whereas DA-VinCi operates nearly twice as fast on UltraScale+ devices with a BRAM speed of 737 MHz. This indicates the potential for even greater performance improvements if DA-VinCi were implemented on Stratix 10.

6.5 Comparison with Custom Accelerators

Outperforming custom accelerators with a more general overlay architecture is very difficult. However, because DA-VinCi achieved the Ideal Design Standards, we expected it to deliver competitive performance compared to custom accelerators. Table 26 compares DA-VinCi's latency and utilization with some state-of-the-art custom deep-learning accelerators implemented on FPGAs. The LSTM-5 benchmark was used to predict the real-time response of high-rate (HRate) dynamic systems on the DROPBEAR dataset in [110]. It consists of an input size of 16 and three hidden layers of size 15. The LSTM-2 benchmark, which is also included in Table 25, has an input size of 64, two hidden layers of size 128, and a fully-connected output layer of size 64. This was used for character recognition on the Shakespeare dataset using a Streaming accelerator (Stream) in [111]. The MLP-2 benchmark, used by CNN-MLP (CM) in [112] for handwritten digit recognition on the MNIST dataset. It includes 784 inputs, two fully-connected layers of size 64, and an output layer of size 10. The GRU-1 benchmark, also included in Table 25, consists of an input size of 39, two hidden layers of sizes 256 and 200, and a fully-connected output layer of size 10. It was used for speech recognition on the DeepSpeech dataset using the DeltaRNN (DRNN) accelerator with an optimal delta threshold of 0x80 in [39]. Lastly, the LSTM-6 benchmark from the DeepBench suite, with an LSTM layer of 1024 hidden units. This was used by the custom accelerator Spartus [40] for speech recognition on the TIMIT dataset.

As observed from Table 26, DA-VinCi achieved up to 40% and 20% of the performance compared to the HLS and HDL implementations of the HRate LSTM-5 accelerator, respectively. DA-VinCi's slower performance in this case is due to its bit-serial computing architecture and the small size of the network. Bit-serial computing is significantly slower than bit-parallel computing, but DA-VinCi compensates for this by providing tens of thousands of bit-serial processing elements. However, because the network is very small in these benchmarks, DA-VinCi loses its parallelism advantage and is hindered by its bit-serial architecture. Despite this, it achieved up to 40% of HRate's performance due to its higher clock speed, at significantly lower resource

Table 27: Application latency comparison of DA-VinCi with the Custom accelerators

Design	Lat.[a]	Speedup	Preci.[b]	LUT	FF	BRAM	DSP	Freq[c]	FPGA[d]	Type
LSTM-5 (16, 15, 15, 15) on DROPBEAR dataset										
HRate [110]	4.72	1	FxP16	25346	31136	16	224	375	US+	HLS
DA-VinCi	11.8	0.4	FxP16	5397	6541	30	15	737	US+	Overlay
HRate [110]	2.49	1	FxP16	65184	130368	41	1174	256	US+	HDL
DA-VinCi	11.8	0.2	FxP16	5397	6541	30	15	737	US+	Overlay
LSTM-2 (64, 128, 128, 64) on CharRec dataset										
Stream [111]	66	1	FxP16	95263	118261	259	1095	420	US+	Overlay
DA-VinCi	18.72	3.5	FxP16	141095	178127	640	128	737	US+	Overlay
MLP-2 (784, 64, 64, 10) on MNIST dataset										
CM [112]	180	1	FxP8	18218	11670	222	6	100	V7	HLS
DA-VinCi	12.2	14.8	FxP8	39524	47700	192	64	540	V7	Overlay
GRU-1 (39, 256, 200, 10) on DeepSpeech dataset										
DRNN [39]	1000[e]	1	FxP16	261357	119260	768	457.5	125	Z7	HDL
DA-VinCi	17.59	56.9	FxP16	534827	692309	2304	256	737	US+	Overlay
LSTM-6 (h=1024) from Deepbench										
Spartus [40]	1	1	INT8	136481	108186	250	520	200	Z7	HDL
DA-VinCi	36.0	0.03	INT8	534827	692309	2304	256	737	US+	Overlay

[a] Execution latency in micro-seconds (us)
[b] Precision FxP := Fixed-Point, INT := Integer
[c] System frequency in MHz
[d] US+ := UltraScale+, V7 := Virtex-7, Z7 := Zynq-7000
[e] The reported latency is for the optimal delta threshold of 0x80 [39]

utilization.

DA-VinCi achieved a 3.5× latency speed-up on the LSTM-2 benchmark compared to the Streaming accelerator [111], which was tailored for the target application using a custom dataflow arrangement and DSP block capabilities directly. Despite the tailored design of the Streaming accelerator, DA-VinCi outperformed it due to its faster clock speed, near-optimal reduction network design, and a higher level of parallelism. Additionally, DA-VinCi demonstrated a substantial 14.8× speed-up on the MLP-2 benchmark compared to CNN-MLP [112] implemented using HLS. DA-VinCi is a software programmable reconfigurable overlay. This speed-up highlights DA-VinCi as a superior choice to HLS-based custom accelerators, offering improved performance without sacrificing the portability, rapid customization, and reconfigurability features of HLS. DA-VinCi also achieved a notable 17.59× speed-up compared to Delta-RNN [39] on the GRU-1 benchmark. Part of this speed-up can be attributed to the faster technology node of UltraScale+ compared to Zynq-7000. It's important to note that Delta-RNN is optimized for exploiting temporal dependencies in RNN inputs and activations, and the latency reported in Table 26 reflects its maximum speed with a delta-threshold of 0x80 [39]. Nevertheless, DA-VinCi significantly outperforms Delta-RNN in application execution latency due to its higher parallelism and roughly 6× faster system frequency.

The inclusion of the LSTM-6 benchmark and comparison with Spartus demonstrates the inherent limitations of overlays compared to highly customized accelerators. Spartus leverages spatio-temporal sparsity through structured column-balanced targeted dropout (CBTD) pruning [40]. It was implemented as a custom accelerator using HDL enabling target platform-specific customizations. In executing the LSTM-6 kernel with sufficient parallelism, Spartus achieved a remarkable 1 us latency, whereas DA-VinCi required 36 us, making DA-VinCi 36 times slower on this benchmark with higher resource utilization. This disparity highlights that tailored accelerators, leveraging application and platform-specific optimizations, can achieve the highest performance at a lower cost. However, such customizations require prolonged design times, lack rapid prototyping capabilities, and often struggle with portability across different platforms or applications. In contrast, DA-VinCi mitigates these drawbacks, delivering competitive or superior performance across a broader range of applications and target platforms.

Chapter 7

Conclusion

This dissertation developed a reconfigurable memory-centric array processor overlay architecture that successfully met a set of Ideal Design Standards. These standards were carefully established by taking into account both analytical reasoning and practical limitations of existing devices. Throughout the development process, the proposed design standards not only served as ambitious yet attainable design goals but also acted as guiding principles to make near-optimal design choices. By adhering to these standards, the dissertation demonstrated how Reconfigurable Memory-Centric Array Processor FPGA Overlays can approach an optimal solution for deep learning applications.

Initially, our research focused on developing PiCaSO, a Processor-in-Memory (PIM) architecture designed to meet stringent Ideal Design Standards. We evaluated PiCaSO across diverse computing platforms, aiming to validate its portability, performance, and scalability. Our comparative studies, which included benchmarking against existing overlay and custom BRAM-based PIM architectures, consistently showcased that PiCaSO can provide higher clock speed and memory efficiency.

Using PiCaSO as the foundation, we developed IMAGine, a high-performance PIM array-based GEMV accelerator. The IMAGine case study demonstrated that achieving the Ideal Design Standard is feasible at the system level. IMAGine showcased its capability to fully exploit the total internal bandwidth of the device operating at the maximum frequency supported by the BRAM. Additionally, scalability studies highlighted that IMAGine's processing capacity scales linearly with increasing BRAM density, even in devices with low LUT-to-BRAM ratios.

In the last chapter, we developed a comprehensive deep-learning accelerator, DA-VinCi, extending IMAGine with support for efficient vector-vector operations and activation functions. Implementation and design analysis showed that DA-VinCi retains all the benefits of IMAGine, while satisfying the complete computing needs of deep-learning applications. Latency and utilization studies were presented between DA-VinCi, SPAR-2, PIM accelerators, and custom FPGA accelerators. The comparative study revealed that DA-VinCi has a significantly faster execution latency compared to existing overlays and custom-BRAM PIM accelerators. It also surpassed many custom FPGA accelerators in terms of latency providing portability and rapid development benefits, making it a better choice given it meets performance requirements within the allocated resources.

The work presented in this study paves the way for several promising avenues of future research. While significant progress has been made in developing the array processor, a notable challenge that persists is ensuring efficient data transfer to sustain its operations. Addressing this challenge involves exploring various strategies such as designing optimal memory hierarchies for the array processors, implementing optimal compression techniques, and developing efficient encoding schemes. These efforts are crucial for maintaining a steady data flow to thousands of processing elements within the array processor, thereby preventing array stalls and reducing latency.

Programming DA-VinCi, like any other novel accelerator, is relatively difficult due to the absence of a standard compiler infrastructure. Overcoming this hurdle requires significant research efforts aimed at refining application mapping techniques that can effectively harness DA-VinCi's computational capabilities. Moreover, DA-VinCi's reconfigurable overlay nature introduces opportunities for innovative software-hardware co-design approaches. This flexibility enables the optimal selection of implementation parameters tailored to specific application requirements, thereby improving both performance and resource utilization. Future advancements in these areas hold the potential to streamline programming workflows and enhance the adaptability of DA-VinCi across a wide range of computational tasks and platforms.

Lastly, novel FPGA device families can be designed taking advantage of the PiCaSO and DA-VinCi architectures. PiCaSO, for instance, offers a compelling solution that can be seamlessly integrated as a custom BRAM block within FPGA designs. This integration not only enhances memory utilization and computing efficiency but also facilitates the deployment of in-memory computing architectures directly on FPGA hardware. Moreover, custom device families, such as AMD's Versal ACAP, exists that are specifically designed to cater to the unique requirements of domain-specific applications. By leveraging DA-VinCi and IMAGine as reference designs, there is a promising opportunity to develop a new family of domain-specific FPGA architectures tailored for memory-intensive applications and in-memory computing paradigms. Incorporating these innovations into future FPGA designs could significantly enhance performance, scalability, and versatility of diverse computational tasks and applications.

References

[1] A. Arora, A. Bhamburkar, A. Borda, T. Anand, R. Sehgal, B. Hanindhito, P.-E. Gaillardon, J. Kulkarni, and L. K. John, “CoMeFa: Deploying Compute-in-Memory on FPGAs for Deep Learning Acceleration,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 3, pp. 1–34, Sep. 2023.

[2] M. A. Kabir, E. Kabir, J. Hollis, E. Levy-Mackay, A. Panahi, J. Bakos, M. Huang, and D. Andrews, “FPGA Processor In Memory Architectures (PIMs): Overlay or Overhaul ?” in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. Gothenburg, Sweden: IEEE, Sep. 2023, pp. 109–115.

[3] A. Panahi, “A memory-centric customizable domain-specific FPGA overlay for accelerating machine learning applications,” Ph.D dissertation, University of Arkansas, 2022.

[4] Y.-k. Choi, K. You, J. Choi, and W. Sung, “A Real-Time FPGA-Based 20 000-Word Speech Recognizer With Optimized DRAM Access,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 57, no. 8, pp. 2119–2131, Aug. 2010.

[5] D. Danopoulos, C. Kachris, and D. Soudris, “Acceleration of image classification with Caffe framework using FPGA,” in *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, May 2018, pp. 1–4.

[6] H. Fan, S. Liu, M. Ferianc, H.-C. Ng, Z. Que, S. Liu, X. Niu, and W. Luk, “A Real-Time Object Detection Accelerator with Compressed SSDLite on FPGA,” in *2018 International Conference on Field-Programmable Technology (FPT)*, Dec. 2018, pp. 14–21.

[7] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, “ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2017, pp. 75–84.

[8] C. Hao, A. Sarwari, Z. Jin, H. Abu-Haimed, D. Sew, Y. Li, X. Liu, B. Wu, D. Fu, J. Gu, and D. Chen, “A Hybrid GPU + FPGA System Design for Autonomous Driving Cars,” in *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*, Oct. 2019, pp. 121–126.

[9] J. P. Klock, J. Corrêa, M. Bessa, J. Arias-Garcia, F. Barboza, and C. Meinertz, “A New Automated Energy Meter Fraud Detection System Based on Artificial Intelligence,” in *2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC)*, Nov. 2021, pp. 1–8.

[10] A. Kojima and Y. Nose, “Development of an Autonomous Driving Robot Car Using FPGA,” in *2018 International Conference on Field-Programmable Technology (FPT)*, Dec. 2018, pp. 411–414.

[11] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung, “FPGA-Based Low-Power Speech Recognition with Recurrent Neural Networks,” in *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, Oct. 2016, pp. 230–235.

[12] P. Lv, W. Liu, and J. Li, “A FPGA-based accelerator implementaion for YOLOv2 object detection using Winograd algorithm,” in *2020 5th International Conference on Mechanical, Control and Computer Engineering (ICMCCE)*, Dec. 2020, pp. 1894–1898.

[13] D. T. Nguyen, T. N. Nguyen, H. Kim, and H.-J. Lee, “A High-Throughput and Power-Efficient FPGA Implementation of YOLO CNN for Object Detection,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1861–1873, Aug. 2019.

[14] S. Sağlam, F. Tat, and S. Bayar, “FPGA Implementation of CNN Algorithm for Detecting Malaria Diseased Blood Cells,” in *2019 International Symposium on Advanced Electrical and Communication Technologies (ISAECT)*, Nov. 2019, pp. 1–5.

[15] D. Selvathi and R. D. Nayagam, “FPGA implementation of on-chip ANN for breast cancer diagnosis,” *Intelligent Decision Technologies*, vol. 10, no. 4, pp. 341–352, Dec. 2016.

[16] D.-F. Syu, S.-W. Syu, S.-J. Ruan, Y.-C. Huang, and C.-K. Yang, “FPGA implementation of automatic speech recognition system in a car environment,” in *2015 IEEE 4th Global Conference on Consumer Electronics (GCCE)*, Oct. 2015, pp. 485–486.

[17] J. Wang and S. Gu, “FPGA Implementation of Object Detection Accelerator Based on Vitis-AI,” in *2021 11th International Conference on Information Science and Technology (ICIST)*, May 2021, pp. 571–577.

[18] H. Xiao, K. Zhao, and G. Liu, “Efficient Hardware Accelerator for Compressed Sparse Deep Neural Network,” *IEICE Transactions on Information and Systems*, vol. E104.D, no. 5, pp. 772–775, May 2021.

[19] A. J. A. El-Maksoud, M. Ebbed, A. H. Khalil, and H. Mostafa, “Power Efficient Design of High-Performance Convolutional Neural Networks Hardware Accelerator on FPGA: A Case Study With GoogLeNet,” *IEEE Access*, vol. 9, pp. 151 897–151 911, 2021.

[20] H. Hu, J. Li, C. Wu, X. Li, and Y. Chen, “Design and Implementation of Intelligent Speech Recognition System Based on FPGA,” *Journal of Physics: Conference Series*, vol. 2171, no. 1, p. 012010, Jan. 2022.

[21] S. Bouguezzi, H. B. Fredj, T. Belabed, C. Valderrama, H. Faiedh, and C. Souani, “An Efficient FPGA-Based Convolutional Neural Network for Classification: Ad-MobileNet,” *Electronics*, vol. 10, no. 18, p. 2272, Sep. 2021.

[22] S. Xiong, G. Wu, X. Fan, X. Feng, Z. Huang, W. Cao, X. Zhou, S. Ding, J. Yu, L. Wang, and Z. Shi, “MRI-based brain tumor segmentation using FPGA-accelerated neural network,” *BMC Bioinformatics*, vol. 22, no. 1, p. 421, Dec. 2021.

[23] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional Architecture for Fast Feature Embedding,” in *Proceedings of the 22nd ACM international conference on Multimedia*, Nov. 2014, pp. 675–678.

[24] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, “High-Performance FPGA-Based CNN Accelerator With Block-Floating-Point Arithmetic,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1874–1885, Aug. 2019.

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

[26] H. Irmak, N. Alachiotis, and D. Ziener, “An Energy-Efficient FPGA-based Convolutional Neural Network Implementation,” in *2021 29th Signal Processing and Communications Applications Conference (SIU)*, Jun. 2021, pp. 1–4.

[27] H. Irmak, F. Corradi, P. Detterer, N. Alachiotis, and D. Ziener, “A Dynamic Reconfigurable Architecture for Hybrid Spiking and Convolutional FPGA-Based Neural Network Designs,” *Journal of Low Power Electronics and Applications*, vol. 11, no. 3, p. 32, Aug. 2021.

[28] H. Khan, A. Khan, Z. Khan, L. B. Huang, K. Wang, and L. He, “NPE: An FPGA-based Overlay Processor for Natural Language Processing,” in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2021, pp. 227–227.

[29] 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 *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, Aug. 2021, pp. 24–27.

[30] K. M. V. Gowda, S. Madhavan, S. Rinaldi, P. B. Divakarachari, and A. Atmakur, “FPGA-Based Reconfigurable Convolutional Neural Network Accelerator Using Sparse and Convolutional Optimization,” *Electronics*, vol. 11, no. 10, p. 1653, May 2022.

[31] A. Panahi, E. Kabir, A. Downey, D. Andrews, M. Huang, and J. D. Bakos, “High-Rate Machine Learning for Forecasting Time-Series Signals,” in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2022, pp. 1–9.

[32] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, “FPGA HLS Today: Successes, Challenges, and Opportunities,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 4, pp. 1–42, Dec. 2022.

[33] Y.-k. Choi, Y. Chi, J. Lau, and J. Cong, “TARO: Automatic Optimization for Free-Running Kernels in FPGA High-Level Synthesis,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2022.

[34] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong, "AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators," *ACM Transactions on Design Automation of Electronic Systems*, vol. 27, no. 4, pp. 1–27, Jul. 2022.

[35] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen, "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Apr. 2022, pp. 741–755.

[36] H. Jun, H. Ye, H. Jeong, and D. Chen, "AutoScaleDSE: A Scalable Design Space Exploration Engine for High-Level Synthesis," *ACM Transactions on Reconfigurable Technology and Systems*, p. 3572959, Feb. 2023.

[37] H. Ye, H. Jun, J. Yang, and D. Chen, "High-level Synthesis for Domain Specific Computing," in *Proceedings of the 2023 International Symposium on Physical Design*, Mar. 2023, pp. 211–219.

[38] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[39] 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*. ACM, Feb. 2018, pp. 21–30.

[40] C. Gao, T. Delbruck, and S.-C. Liu, "Spartus: A 9.4 TOp/s FPGA-based LSTM accelerator exploiting spatio-temporal sparsity," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2022.

[41] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "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)*, Apr. 2017, pp. 152–159.

[42] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs," in *Proceedings of the 54th Annual Design Automation Conference 2017*, Jun. 2017, pp. 1–6.

[43] M. S. Abdelfattah, D. Han, A. Bitar, R. DiCecco, S. O'Connell, N. Shanker, J. Chu, I. Prins, J. Fender, A. C. Ling, and G. R. Chiu, "DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2018, pp. 411–4117.

[44] Y. Chi, J. Cong, P. Wei, and P. Zhou, "SODA: Stencil with Optimized Dataflow Architecture," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2018, pp. 1–8.

[45] J. Cong and J. Wang, "PolySA: Polyhedral-Based Systolic Array Auto-Compilation," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2018, pp. 1–8.

[46] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Latte: Locality Aware Transformation for High-Level Synthesis," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2018, pp. 125–128.

[47] C. H. Yu, P. Wei, M. Grossman, P. Zhang, V. Sarker, and J. Cong, "S2FA: an accelerator automation framework for heterogeneous computing in datacenters," in *Proceedings of the 55th Annual Design Automation Conference*, Jun. 2018, pp. 1–6.

[48] 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)*, Nov. 2018, pp. 1–8.

[49] 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, Nov. 2019.

[50] Y. Chen, K. Zhang, C. Gong, C. Hao, X. Zhang, T. Li, and D. Chen, "T-DLA: An Open-source Deep Learning Accelerator for Ternarized DNN Models on Embedded FPGA," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Jul. 2019, pp. 13–18.

[51] J. Li, Y. Chi, and J. Cong, "HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2020, pp. 51–57.

[52] J. Wang, L. Guo, and J. Cong, "AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2021, pp. 93–104.

[53] S. Basalama, A. Sohrabizadeh, J. Wang, L. Guo, and J. Cong, "FlexCNN: An End-to-End Framework for Composing CNN Accelerators on FPGA," *ACM Transactions on Reconfigurable Technology and Systems*, p. 3570928, Dec. 2022.

[54] S. Liu, J. Weng, D. Kupsh, A. Sohrabizadeh, Z. Wang, L. Guo, J. Liu, M. Zhulin, R. Mani, L. Zhang, J. Cong, and T. Nowatzki, "OverGen: Improving FPGA Usability through Domain-specific Overlay Generation," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2022, pp. 35–56.

[55] J. Zhuang, J. Lau, H. Ye, Z. Yang, Y. Du, J. Lo, K. Denolf, S. Neuendorffer, A. Jones, J. Hu, D. Chen, J. Cong, and P. Zhou, "CHARM: Composing Heterogeneous Accelerators for Matrix Multiply on Versal ACAP Architecture," Jan. 2023.

[56] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: a DRAM-based Reconfigurable In-Situ Accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. Cambridge Massachusetts: ACM, Oct. 2017, pp. 288–301.

[57] S. Li, A. O. Glova, X. Hu, P. Gu, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "SCOPE: A Stochastic Computing Engine for DRAM-Based In-Situ Accelerator," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2018, pp. 696–709.

[58] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang, "DrAcc: a DRAM based accelerator for accurate CNN inference," in *Proceedings of the 55th Annual Design Automation Conference*. San Francisco California: ACM, Jun. 2018, pp. 1–6.

[59] Q. Deng, Y. Zhang, M. Zhang, and J. Yang, "LAcc: Exploiting Lookup Table-based Fast and Accurate Vector Multiplication in DRAM-based CNN Accelerator," in *Proceedings of the 56th Annual Design Automation Conference 2019*. Las Vegas NV USA: ACM, Jun. 2019, pp. 1–6.

[60] J. D. Ferreira, G. Falcao, J. Gómez-Luna, M. Alser, L. Orosa, M. Sadrosadati, J. S. Kim, G. F. Oliveira, T. Shahroodi, A. Nori, and O. Mutlu, "pLUTo: Enabling Massively Parallel Computation in DRAM via Lookup Tables," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2022, pp. 900–919.

[61] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2018, pp. 383–396.

[62] C. Eckert, A. Subramaniyan, X. Wang, C. Augustine, R. Iyer, and R. Das, "Eidetic: An In-Memory Matrix Multiplication Accelerator for Neural Networks," *IEEE Transactions on Computers*, pp. 1–14, 2022.

[63] M. Ali, A. Jaiswal, S. Kodge, A. Agrawal, I. Chakraborty, and K. Roy, "IMAC: In-Memory Multi-Bit Multiplication and ACcumulation in 6T SRAM Array," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 8, pp. 2521–2531, Aug. 2020.

[64] S. Yin, Z. Jiang, J.-S. Seo, and M. Seok, "XNOR-SRAM: In-Memory Computing SRAM Macro for Binary/Ternary Deep Neural Networks," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 6, pp. 1733–1743, Jun. 2020.

[65] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. Seoul, South Korea: IEEE, Jun. 2016, pp. 27–39.

[66] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with

In-Situ Analog Arithmetic in Crossbars," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. Seoul, South Korea: IEEE, Jun. 2016, pp. 14–26.

[67] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2017, pp. 541–552.

[68] T. Tang, L. Xia, B. Li, Y. Wang, and H. Yang, "Binary convolutional neural network on RRAM," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2017, pp. 782–787.

[69] X. Qiao, X. Cao, H. Yang, L. Song, and H. Li, "Atomlayer: a universal reRAM-based CNN accelerator with atomic layer computation," in *Proceedings of the 55th Annual Design Automation Conference*. San Francisco California: ACM, Jun. 2018, pp. 1–6.

[70] X. Sun, S. Yin, X. Peng, R. Liu, J.-s. Seo, and S. Yu, "XNOR-RRAM: A scalable and parallel resistive synaptic architecture for binary neural networks," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2018, pp. 1423–1428.

[71] C.-X. Xue, Y.-C. Chiu, T.-W. Liu, T.-Y. Huang, J.-S. Liu, T.-W. Chang, H.-Y. Kao, J.-H. Wang, S.-Y. Wei, C.-Y. Lee, S.-P. Huang, J.-M. Hung, S.-H. Teng, W.-C. Wei, Y.-R. Chen, T.-H. Hsu, Y.-K. Chen, Y.-C. Lo, T.-H. Wen, C.-C. Lo, R.-S. Liu, C.-C. Hsieh, K.-T. Tang, M.-S. Ho, C.-Y. Su, C.-C. Chou, Y.-D. Chih, and M.-F. Chang, "A CMOS-integrated compute-in-memory macro based on resistive random-access memory for AI edge devices," *Nature Electronics*, vol. 4, pp. 81–90, Jan. 2021.

[72] W. Wan, R. Kubendran, C. Schaefer, S. B. Eryilmaz, W. Zhang, D. Wu, S. Deiss, P. Raina, H. Qian, B. Gao, S. Joshi, H. Wu, H.-S. P. Wong, and G. Cauwenberghs, "A compute-in-memory chip based on resistive random-access memory," *Nature*, vol. 608, pp. 504–512, Aug. 2022.

[73] D. Fan and S. Angizi, "Energy Efficient In-Memory Binary Deep Neural Network Accelerator with Dual-Mode SOT-MRAM," in *2017 IEEE International Conference on Computer Design (ICCD)*, Nov. 2017, pp. 609–612.

[74] S. Angizi, Z. He, F. Parveen, and D. Fan, "IMCE: Energy-efficient bit-wise in-memory convolution engine for deep neural network," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2018, pp. 111–116.

[75] Y. Pan, P. Ouyang, Y. Zhao, W. Kang, S. Yin, Y. Zhang, W. Zhao, and S. Wei, "A Multilevel Cell STT-MRAM-Based Computing In-Memory Accelerator for Binary Convolutional Neural Network," *IEEE Transactions on Magnetics*, vol. 54, no. 11, pp. 1–5, Nov. 2018.

[76] A. D. Patil, H. Hua, S. Gonugondla, M. Kang, and N. R. Shanbhag, "An MRAM-Based Deep In-Memory Architecture for Deep Neural Networks," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2019, pp. 1–5.

[77] S. Angizi, Z. He, A. S. Rakin, and D. Fan, "CMP-PIM: an energy-efficient comparator-based processing-in-memory neural network accelerator," in *Proceedings of the 55th Annual Design Automation Conference*. San Francisco California: ACM, Jun. 2018, pp. 1–6.

[78] S. Angizi, Z. He, A. Awad, and D. Fan, "MRIMA: An MRAM-Based In-Memory Accelerator," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 5, pp. 1123–1136, May 2020.

[79] X. Wang, V. Goyal, J. Yu, V. Bertacco, A. Boutros, E. Nurvitadhi, C. Augustine, R. Iyer, and R. Das, "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)*, May 2021, pp. 88–96.

[80] A. Arora, T. Anand, A. Borda, R. Sehgal, B. Hanindhito, J. Kulkarni, and L. K. John, "CoMeFa: Compute-in-Memory Blocks for FPGAs," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. New York City, NY, USA: IEEE, May 2022, pp. 1–9.

[81] Y. Chen and M. S. Abdelfattah, "BRAMAC: Compute-in-BRAM Architectures for Multiply-Accumulate on FPGAs," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Marina Del Rey, CA, USA: IEEE, May 2023, pp. 52–62.

[82] Y. Chen, J. Dotzel, and M. S. Abdelfattah, "M4BRAM: Mixed-Precision Matrix-Matrix Multiplication in FPGA Block RAMs," Nov. 2023, arXiv:2311.02758 [cs]. [Online]. Available: http://arxiv.org/abs/2311.02758

[83] A. Arora, T. Anand, A. Borda, R. Sehgal, B. Hanindhito, J. Kulkarni, and L. K. John, "CoMeFa: Compute-in-Memory Blocks for FPGAs," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2022, pp. 1–9.

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

[85] *Virtex-7 T and XT FPGAs Data Sheet: DC and AC Switching Characteristics*, AMD, 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds183_Virtex_7_Data_Sheet

[86] *Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics*, AMD, 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds923-virtex-ultrascale-plus

[87] *Intel® Stratix® 10 Device Datasheet*, Intel, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683181/current/memory-block-specifications.html

[88] *UltraScale Architecture Libraries Guide*, AMD, 2021. [Online]. Available: https://docs.xilinx.com/v/u/2018.1-English/ug974-vivado-ultrascale-libraries

[89] *Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891)*, AMD, 2022. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview

[90] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural Cache: Bit-Serial in-Cache Acceleration of Deep Neural Networks," in *2018 ACM/IEEE 45Th annual international symposium on computer architecture (ISCA)*, 2018, pp. 383–396.

[91] *UltraFast Design Methodology Guide for FPGAs and SoCs (UG949)*, AMD, 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug949-vivado-design-methodology/Control-Signals-and-Control-Sets

[92] Ling Zhuo, G. Morris, and V. Prasanna, "Designing Scalable FPGA-Based Reduction Circuits Using Pipelined Floating-Point Cores," in *19th IEEE International Parallel and Distributed Processing Symposium*, 2005, pp. 147a–147a.

[93] L. Zhuo, G. R. Morris, and V. K. Prasanna, "High-Performance Reduction Circuits Using Deeply Pipelined Operators on FPGAs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 10, pp. 1377–1392, 2007.

[94] Y.-G. Tai, C.-T. D. Lo, and K. Psarris, "Accelerating Matrix Operations with Improved Deeply Pipelined Vector Reduction," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 202–210, 2012.

[95] M. Huang and D. Andrews, "Modular Design of Fully Pipelined Reduction Circuits on FPGAs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 9, pp. 1818–1826, 2013.

[96] L. Tang, G. Cai, Y. Zheng, and J. Chen, "A Resource and Performance Optimization Reduction Circuit on FPGAs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 355–366, 2021.

[97] S. Ullah, S. Rehman, M. Shafique, and A. Kumar, "High-Performance Accurate and Approximate Multipliers for FPGA-Based Hardware Accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 2, pp. 211–224, 2022.

[98] M. A. Kabir, E. Kabir, J. Hollis, E. Levy-Mackay, A. Panahi, J. Bakos, M. Huang, and D. Andrews, "PiCaSO: A Scalable and Fast PIM Overlay," 2023. [Online]. Available: https://github.com/Arafat-Kabir/PiCaSO

[99] X. Wang, V. Goyal, J. Yu, V. Bertacco, A. Boutros, E. Nurvitadhi, C. Augustine, R. Iyer, and R. Das, "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)*, May 2021, pp. 88–96.

[100] *UltraFast Design Methodology Guide for FPGAs and SoCs (UG949)*, AMD. [Online]. Available: https://docs.xilinx.com/r/en-US/ug949-vivado-design-methodology/Reducing-Control-Sets

[101] M. A. Kabir, T. Kamucheka, N. Fredricks, J. Mandebi, J. Bakos, M. Huang, and D. Andrews, "IMAGine: An In-Memory Accelerated GEMV Engine Overlay," 2024. [Online]. Available: https://github.com/Arafat-Kabir/IMAGine

[102] *Alveo U55C Data Center Accelerator Card User Guide*, AMD, 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1469-alveo-u55c

[103] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten lessons from three generations shaped google's TPUv4i : Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Jun. 2021, pp. 1–14.

[104] Y. Jiao, L. Han, R. Jin, Y.-J. Su, C. Ho, L. Yin, Y. Li, L. Chen, Z. Chen, L. Liu, Z. He, Y. Yan, J. He, J. Mao, X. Zai, X. Wu, Y. Zhou, M. Gu, G. Zhu, R. Zhong, W. Lee, P. Chen, Y. Chen, W. Li, D. Xiao, Q. Yan, M. Zhuang, J. Chen, Y. Tian, Y. Lin, W. Wu, H. Li, and Z. Dou, "7.2 a 12nm programmable convolution-efficient neural-processing-unit chip achieving 825tops," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*. IEEE, Feb. 2020, pp. 136–140.

[105] J. W. Lyons, "DARPA TIMIT acoustic-phonetic continuous speech corpus," *National Institute of Standards and Technology*, 1993.

[106] [Online]. Available: https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt

[107] Y. LeCun, C. Cortes, C. Burges *et al.*, "MNIST handwritten digit database," 2010.

[108] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates *et al.*, "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[109] Baidu Research, "Baidu DeepBench." [Online]. Available: https://svail.github.io/DeepBench/

[110] E. Kabir, D. Coble, J. N. Satme, A. R. Downey, J. D. Bakos, D. Andrews, and M. Huang, "Accelerating LSTM-based high-rate dynamic system models," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, Sep. 2023, pp. 327–332.

[111] L. Ioannou and S. A. Fahmy, "Streaming Overlay Architecture for Lightweight LSTM Computation on FPGA SoCs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 1, pp. 1–26, Mar. 2023.

[112] E. Kabir, A. Poudel, Z. Aklah, M. Huang, and D. Andrews, “A runtime programmable accelerator for convolutional and multilayer perceptron neural networks on FPGA,” in *International Symposium on Applied Reconfigurable Computing*. Springer, Oct. 2022, pp. 32–46.