DA-VinCi: A Deep-Learning Accelerator Overlay Using In-Memory Computing

MD ARAFAT KABIR and NATHANIEL FREDRICKS, University of Arkansas, Fayetteville, Arkansas, USA
TENDAYI KAMUCHEKA, University of Arkansas Fayetteville, Fayetteville, Arkansas, USA
JOEL MANDEBI, Advanced Micro Devices, Inc., (AMD), Santa Clara, California, USA
MIAOQING HUANG, University of Arkansas, Fayetteville, Arkansas, USA
JASON D. BAKOS, University of South Carolina, Columbia, South Carolina, USA
DAVID ANDREWS, University of Arkansas, Fayetteville, Arkansas, USA

The matrix operations that underpin today's deep learning models are routinely implemented in Single Instruction Multiple Data (SIMD) domain specific accelerators. SIMD accelerators including GPUs and array processors can effectively leverage parallelism in models that are compute-bound, but their effectiveness can be diminished for models that are memory-bound. Processing-in-Memory (PIM) architectures are being explored to provide better energy efficiency and scalable performance for these memory-bound models. Modern Field Programmable Gate Arrays (FPGAs) feature hundreds of megabits of Static Random Access Memory (SRAM) distributed across the device as disaggregated memory resources. This makes FPGAs ideal programmable platforms for developing custom Processor In/Near Memory accelerators. Several PIM array-based accelerator designs have been proposed to leverage this substantial internal bandwidth. However, results reported to date show the FPGA based PIM architectures operating at system clock frequencies well below a chips Block-RAM (BRAM) Fmax clock frequency. Results also show that the compute densities of the designs do not scale linearly with BRAM densities. These results indicate that FPGA PIM architectures will never be competitive with their custom Application-Specific Integrated Circuit (ASIC) counterparts.

In this article, we introduce DA-VinCi, a *D*eep-Learning *A*ccelerator O*v*erlay using *In*-Memory *C*omput*i*ng. DA-VinCi is the first scalable FPGA based PIM deep-learning accelerator overlay capable of clocking at the maximum frequency of a device's BRAM. Further, the architecture of DA-VinCi allows the number of compute units to scale linearly up to the maximum capacity of a devices BRAM, and at the maximum clock frequency of the BRAM. The DA-VinCi overlay has a programmable Instruction Set Architecture (ISA) that allows the same synthesized design to provide low-latency inferencing of a range of memory-bound deep-learning models, including Multilayer Perceptrons, Recurrent Neural Network, Long Short-Term Memory, and Gated Recurrent Unit networks. The scalability and high clocking frequency of DA-VinCi is achieved through a new Processor In Memory (PIM) tile architecture and a highly scalable system-level framework. We present results showing DA-VinCi linearly scaling the number of Processing Elements (PEs) to 100% of the BRAM capacity (over

This material is based upon work partially supported by the National Science Foundation under Grant No. 1955820.
Authors' Contact Information: MD Arafat Kabir (corresponding author), University of Arkansas, Fayetteville, Arkansas, USA; e-mail: arafat.sun@gmail.com; Nathaniel Fredricks, University of Arkansas, Fayetteville, Arkansas, USA; e-mail: njfredri@uark.edu; Tendayi Kamucheka, University of Arkansas Fayetteville, Fayetteville, Arkansas, USA; e-mail: tfkamuch@uark.edu; Joel Mandebi, Advanced Micro Devices, Inc., (AMD), Santa Clara, California, USA; e-mail: jmandebi@amd.com; Miaoqing Huang, University of Arkansas, Fayetteville, Arkansas, USA; e-mail: mqhuang@uark.edu; Jason D. Bakos, University of South Carolina, Columbia, South Carolina, USA; e-mail: jbakos@cse.sc.edu; David Andrews, University of Arkansas, Fayetteville, Arkansas, USA; e-mail: dandrews@uark.edu.

ACM 1936-7414/2025/11-ART46
https://doi.org/10.1145/3770756

60K PEs) on an Alveo U55 clocking at 737 MHz, the chips BRAM Fmax. We provide comparative studies on inference latency across multiple deep-learning applications that show DA-VinCi achieves up to a 201× improvement over a state-of-the-art PIM overlay accelerator, up to 87× improvement over existing PIM-based FPGA accelerators, and up to 57× improvement over custom deep-learning accelerators on FPGAs.

CCS Concepts: • **Computer systems organization** → **Single instruction, multiple data**; **Systolic arrays**;

Additional Key Words and Phrases: Processing-in-Memory, Deep-learning, FPGA Accelerator, Block RAM, GEMV engine, Processor Array

ACM Reference format:
MD Arafat Kabir, Nathaniel Fredricks, Tendayi Kamucheka, Joel Mandebi, Miaoqing Huang, Jason D. Bakos, and David Andrews. 2025. DA-VinCi: A Deep-Learning Accelerator Overlay Using In-Memory Computing. *ACM Trans. Reconfig. Technol. Syst.* 18, 4, Article 46 (November 2025), 38 pages.
https://doi.org/10.1145/3770756

1 Introduction

Deep Neural Networks (DNNs) are widely used in AI applications ranging from computer vision and voice recognition, through autonomous driving vehicles and robotics, and into large language models and generative AI applications. In some cases, the performance and accuracy of a DNN can exceed that of human intelligence. However, these advantages come at a cost. The size of the datasets needed to achieve the desired accuracy in generative AI models is growing exponentially. The associated computational costs required to process these datasets are accelerating at rates not seen in the past. One recent estimate predicted the next generation of large language models will require 740× percent increase in compute in 2 years.

The high compute requirements of today's AI models are addressed with **Single Instruction Multiple Data (SIMD)** accelerators for both offline training as well as the user-facing inferencing step [1–19]. While SIMD accelerators such as GPUs and array processors can efficiently leverage parallelism in compute-bound deep-learning models, they can become less efficient when facing memory or I/O bound models. A study by Google on their **Tensor Processing Unit (TPU)** servers [34] highlighted this disparity. Compute-bound **Convolutional Neural Networks (CNNs)** kept the compute array active for 78.2% of the total execution cycles. In contrast the more memory-bound **Multilayer Perceptrons (MLPs)** and **Long Short-Term Memory (LSTM)** networks kept the compute array busy for only 10% of the execution cycles, with the array idling over 80% of the total execution time waiting for data or weights to be loaded from memory. These memory and I/O bottleneck problems [20–33] have motivated the re-exploration of **Processor-in-Memory (PIM)** style architectures. PIM architectures integrate processors within the memory system, enabling computations to be performed on data in place. This eliminates the Von Neumann bottleneck and has the potential to linearly scale processing performance with memory capacity.

Modern Field Programmable Gate Arrays (FPGAs) feature hundreds of megabits of Static Random Access Memory (SRAM) distributed across the device as disaggregated memory resources offering terabytes per second of internal bandwidth. The FPGA's **Lookup Tables (LUTs)** and diffused **Multiply-Accumulate (MAC)** IPs can be configured with the **Block-RAMs (BRAMs)** to form PIM tiles. This renders FPGAs potentially ideal programmable platforms for developing custom Processor In/Near Memory accelerators. Unfortunately, the performance results for FPGA-based PIM array accelerators reported in the literature [25–27] show a degradation of the BRAM

clock frequencies and lower than ideal compute densities. Degraded BRAM clock frequencies have also plagued efforts to re-architect an FPGA's separate LUT and BRAM components into a new reconfigurable fabric [24, 28–31]. This led one research effort to opine that accepting a reduced BRAM Fmax may be a characteristic of the PIM approach [24]. If true, this would render the use of PIM FPGAs non-competitive with custom **Application-Specific Integrated Circuit (ASICs)**.

The majority of FPGA based PIM array research to date has focused on the construction of individual PIM tiles, and has not addressed how to efficiently integrate thousands of individual PIM tiles into an integrated larger system. These missing system-level integration studies are critical in understanding if PIM architectures, regardless of achievable Fmax, can realistically exploit an FPGA's full internal BRAM bandwidth. Additional limitations of these efforts include a lack of software development environments which ultimately limits their use to researchers with detailed knowledge of hardware design.

This article presents a treatise of our work exploring the two fundamental questions raised above concerning the relevance of future FPGA-based PIM architectures. Specifically, (1) can FPGA PIM tiles run at the BRAM Fmax, and (2) can compute scale linearly up to a chip's full BRAM bandwdith. The groundwork for addressing the first question was laid in our prior work called ***P*rocessor *i*n/near Memory S*ca*lable and Fa*s*t *O*verlay (PiCaSO)** architecture and IMAGine. PiCaSO presented architecture explorations of 4×4 PIM tiles required to clock at a BRAMs Fmax [32, 33]. IMAGine used PiCaSO tiles to form GEMV tiles [35, 36]. This article extends our prior work to then address the second question concerning the linear scalability of PIM architectures up to an FPGA's full BRAM bandwidth. The outcome of this work is ***D*eep-Learning *A*ccelerator O*v*erlay Using *In*-Memory *C*omput*i*ng (DA-VinCi)**. DA-VinCi is a parameterizable overlay allowing portability studies to be conducted across a range of AMD logic families. Results are provided that analyze DA-VinCi on devices with a range of LUT to BRAM ratios. DA-VinCi includes a complete **Instruction Set Architecture (ISA)** as well as a set of domain specific macro instructions, and an assembler/compiler toolchain. As such, we believe DA-VinCi is the first open source end-to-end PIM overlay + software environment freely available to researchers and machine learning domain experts.

A key contribution of our work is the identification and application of design techniques that enable FPGA-based PIM architectures to scale compute linearly with the full BRAM bandwidth of a device, while operating at the BRAM Fmax. We show that these design techniques are equally applicable to overlays as well as new PIM-based reconfigurable fabrics. The performance analysis presented in Section 8.2 shows DA-VinCi achieves up to a 201× improvement over a baseline PIM overlay accelerator, up to 87× improvement over existing custom-BRAM PIM-based FPGA accelerators, and up to 57× improvement over custom deep-learning accelerators on FPGAs. We present results that establish DA-VinCi as the fastest PIM-based GEMV overlay, outperforming even the custom PIM-based FPGA accelerators reported to date. Additionally, it surpasses TPU v1-v2 and Alibaba Hanguang 800 in clock speed while offering an equal or greater number of MAC units.

The contributions of this article can be summarized as follows:

- A PIM overlay architecture, DA-VinCi, that achieves BRAM-level Fmax and scales compute units to utilize 100% of the available BRAM capacity across AMD FPGA families.
- An in-depth analysis of design tradeoffs that enable high-frequency and highly scalable PIM implementations using on-chip BRAMs.
- A novel **Vector–Vector Engine (VV-Engine)** that complements GEMV computation by accelerating post-GEMV point-wise operations in deep-learning applications.

— A light-weight and open-sourced programming framework for deploying machine learning models on DA-VinCi overlays, including a Python-based assembler for ISA-level code generation.
— A simplified C-compatible programming flow and runtime interface for system-level integration of DA-VinCi as a co-processor.
— An experimental evaluation showing that DA-VinCi achieves up to two orders of magnitude improvement in inference latency compared to existing PIM overlays and custom FPGA accelerators.

DA-VinCi is available at [37] as open source implementation and is freely available for study, use, modification, and distribution without restriction.

2 Related Work

2.1 PIM Architectures

Processor in/near memory Architectures are not new, with origins dating back to the 1969 Cellular Logic-in-Memory Array by Kautz [38] and the 1970 Logic-in-Memory Computer by Stone [39]. Kautz proposed a 2D array of identical elementary cells each with a small amount of logic and storage such that the same array could be regarded as either a logically enhanced memory array or as a logic array that could be programmed to realize a desired logical behavior. Stone envisioned placing logic within a cache that could perform arithmetic and logical operations between elements of two cache blocks as well as perform associative operations within a cache block. Stone posited that the cache would give the illusion to the processor that operations were occurring throughout the main memory but at the speed of the cache.

The very definition of the term "**Processor in Memory (PIM)**" has evolved throughout the years since these early works, and now serves as an umbrella term that covers two unique approaches. The first approach termed Processing Using Memory refers to architectures that enable the memory cells themselves to compute. This approach is a modern-day extension of Kuatz's original Cellular Logic-in-Memory Array. These architectures are gaining renewed interest as we enter the post Moore's Law generation. Prototype CMOS+Memristor hybrid architectures that include 2D matrices of analog memristors have shown up to 10,000 TOPS/Watt efficiencies running GEMV applications. This level of performance efficiency makes the approach attractive for implementing the matrix operations that dominate todays AI models. Further, the emerging non-volatile memory nanotechnologies used as compute memory cells are also a promising solution to handle the exponential growth in memory storage and data bandwidth requirements [40–42].

The second approach termed "**Processor near Memory (PNM)**" refers to architectures that place **Processing Elements (PEs)** within close proximity to, but not integrated within, the memory cell itself. The nearness of the PEs to the memory can vary based on how the system is designed as well as what technologies are used. Our work and the remainder of this article focuses on this PNM approach. To maintain consistency with the existing literature we will use the term PIM to refer to PNM. The model of computation for a PIM architecture generally includes small bit-serial PE's residing near the ends of a memory bank's bit lines [43]. The PE's function as a SIMD data-parallel array processor with the level of concurrency only limited by the memory bandwidth.

Figure 1 from Gokhale et al. [44] show an early Terasys PIM tile that integrated 64 bit serial ALUs with a 2Kbit (32K × 4 bit) SRAM memory bank. Elliott's computational RAM (C*RAM) [45] is yet another early representative example of a PIM architecture that targeted the use of DRAM technology. A prototype 64 PE C-RAM chip was fabricated in SRAM. A follow-on 2048 PE 4Mbit DRAM chip was designed but not fabricated. These early PIM architectures were developed to accelerate applications such as image processing, discrete event simulations including computational

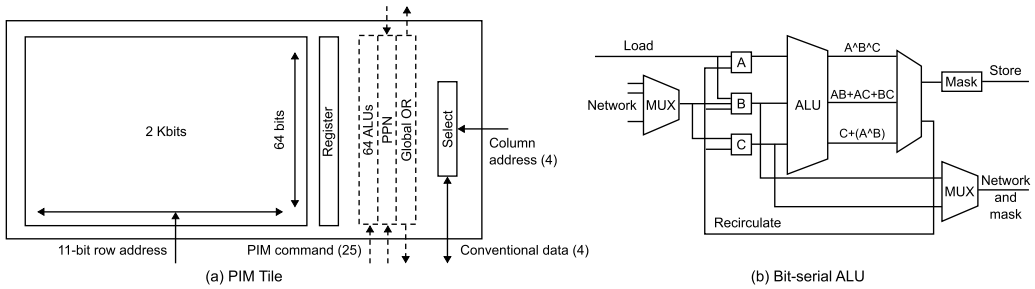

Fig. 1. Teraysis PIM architecture [44].

fluid dynamics, DNA sequencing, and matrix operations that exhibited largely independent data parallelism and nearest-neighbor communications. Nearest neighbor **North-East-West-South (NEWS)** networks between the PIM tiles were needed to support the nearest neighbor data transfer patterns of the target applications. The Berkeley IRAM project [46–48] is yet another example of a PIM architecture that targeted fabrication in a memory fabrication process.

The proliferation of machine learning models has renewed interest in PIM architectures. Tightly integrating bit serial PE's with distributed banks of memory can eliminate the classic von Neumann bottleneck and reduce the large energy drain associated with transferring data between separate memory and processor chips. Ideally a PIM architecture will scale the number of PE's linearly with the available on-chip internal memory bandwidth. In general this makes them particularly efficient at exploiting the data level parallelism of the matrix operations that underpin deep learning models. Matching PE's to the memory bandwidth is particularly promising for models that are memory-bound with moderate to low operational intensity.

Commercial Processor in/near Memory architectures have recently become available. UPMEM became the first commercially available PIM hardware [49, 50]. Other companies have since followed suit. Samsung developed the HBM2 Aquabolt PIM to improve performance of bandwidth-intensive workloads and improve energy efficiency by reducing computing-memory data movement [51].

2.2 FPGA PIM Accelerators

The benefits of PIM architectures are also being explored within the reconfigurable computing community. Investigations of new reconfigurable fabrics that integrate traditional bit-serial arithmetic units into an FPGAs BRAM IP to form a PIM tile. This trend is relatively recent in the FPGA community with only a few works published to date. Inspired by Neural Cache [52], Wang et al. [53] proposed **Compute-Capable BRAM (CCB)** that added in-memory compute capabilities to FPGA BRAMs. The BRAMs could act as storage units or execute bit-serial arithmetic operations. Results showed a 1.6× and 2.3× increase in peak MA 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 based on the CCB design for deep-learning inference. RIMA used the CCBs and the FPGA's reconfigurability to achieve 1.25× and 3× higher performance on average for 8-bit integer and block floating-point precisions, respectively, compared to the state-of-the-art Brainwave deep-learning soft processor. Results showed RIMA implemented within a Stratix 10 FPGA could achieve an order of magnitude higher performance than a same-generation GPU.

Improving upon CCB, Arora et al. [54] proposed modification to FPGA BRAMs to create **Compute-in-Memory Blocks for FPGAs (CoMeFa)** RAMs, which provided 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 bit-serial PEs. They can be used for computing with any precision, making them valuable for deep-learning applications. Two variations were proposed: CoMeFa-D optimized for delay, and CoMeFa-A optimized for area. By adding CoMeFa-D or CoMeFa-A RAMs to an Intel Arria-10-like FPGA, a geomean speedup of 2.55× or 1.85× was shown, 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 MAC dataflow in BRAMAC [30] and M4BRAM [31]. 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, it copies 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. Combining BRAMAC-2SA/BRAMAC-1DA with Intel's DLA [55] resulted in an average speed-up of 2.05×/1.7× for AlexNet and 1.33×/1.52× for ResNet-34. M4BRAM surpassed BRAMAC by an average of 1.43× across diverse benchmarks.

2.3 FPGA-Based Custom Accelerators

We wrap up the background section with descriptions of the FPGA-based custom Accelerators used in our comparative studies. While there are many FPGA-based custom accelerators appearing in the literature, these specific custom accelerators were chosen by their high relevance with the deep learning models we used as benchmarks to evaluate Da-VinCi. Each of these custom FPGA-based accelerators were tailored to a specific application with the design goal of delivering optimal performance and power efficiency at a minimal cost. This makes them good benchmark systems in which to evaluate performance losses that may be incurred with a programmable overlay such as DA-VinCi.

Gao et al. [56] proposed a **Gated Recurrent Unit–Recurrent Neural Network (GRU-RNN)** accelerator architecture called **Delta-RNN (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 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× speedup compared to a conventional RNN update because of the sparsity created by the Delta Network algorithm and the zero-skipping ability of DRNN.

Later, they proposed Spartus [57] improving upon Delta-RNN, 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 1 µs for a single Delta-LSTM layer of 1,024 neurons. It achieved 46× speed-up for the LSTM network using the TIMIT dataset.

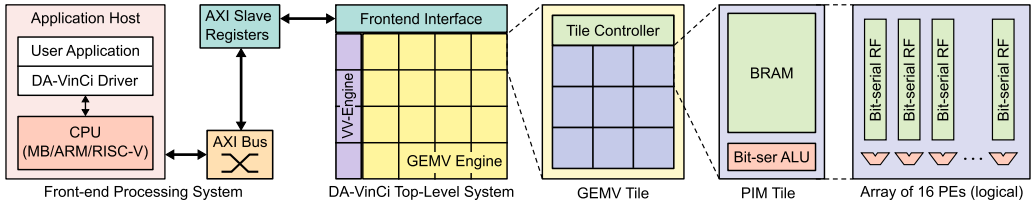

Fig. 2. End-to-end organization and component hierarchy of a system using DA-VinCi as a co-processor through AXI bus. On Alveo U55 board, DA-VinCi delivers 60K PEs hierarchically organized into a 2D grid of GEMV tiles, each containing 12×2 array of PIM tiles. Each PIM tile is logically organized as a 1×16 array of bit-serial PEs.

Kabir et al. [58] proposed **CNN-MLPA (CM)**, an **High-Level Synthesis (HLS)**-based runtime programmable accelerator overlay for FPGAs. The objective of the work was to create a single software programmable accelerator that could run both CNNs and MLPs of any topology without resynthesizing for different networks. The design partitions neurons within the network into groups of available PEs. This showed improved reusability and scalability of a programmable accelerator over custom accelerators. CM was implemented on a Virtex-7 FPGA, which achieved a throughput of 610 MOP/S running at 100 MHz and a significantly better normalized throughput compared to many state-of-the-art deep learning accelerators.

Later, the same authors proposed a custom LSTM accelerator on FPGA [59]. Two versions were created: one through HLS and the second using a **Hardware Description Language (HDL)**. The focus of the design was the **Dynamic Reproduction of Projectiles in Ballistic Environments for Advanced Research (DROPBEAR)** dataset [60], which was generated as a benchmark for the study of real-time structural modeling applications. The LSTM accelerator was optimized specifically for latency with the HDL version outperforming the HLS version achieving an inference latency of 1.42 µs and throughput of 7.87 GOP/s on an Alveo U55C platform.

Ioannou and Fahmy [61] proposed a flexible overlay architecture for LSTMs on FPGA SoCs. This architecture was built around a streaming dataflow arrangement and directly utilized DSP block capabilities. It was designed to keep parameters within the architecture while serially moving input data to reduce external memory access overhead. It achieved 82.6% of the theoretical DSP block maximum frequency and 22.6× more efficiency compared to the average performance of the existing designs.

3 DA-VinCI Architecture Overview

DA-VinCi has been designed to operate as a standard domain-specific co-processor accelerator, similarly to Google's TPU [34]. As shown in Figure 2, a front-end processor sets up a machine learning model by first transferring the weight matrix from DRAM into the local distributed BRAMs within the PIM array. The size of the weight matrix that can be resident at any one time in the local memory is equal to an FPGA's BRAM capacity. Matrices larger than the BRAM capacity are decomposed into submatrices with weight transfers between DRAM and BRAM, as well as execution controlled by the front-end processor. The front-end processor issues co-processor instructions from cross-compiled kernels or on-the-fly driver-generated instructions through a memory-mapped AXI interface. A standard ISA has been defined and is presented in Section 7.2. DA-VinCi executes instructions decoupled from the front-end processor. This allows the execution of variable-precision multicycle bit-serial arithmetic operations.

Figure 2 breaks down DA-VinCi's top-level system architecture into three subsystems: a modular and parameterizable GEMV engine, a VV-Engine, and a Front-end Interface. The modular GEMV

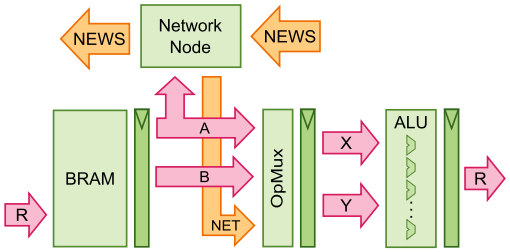

Fig. 3. Datapath of the proposed PIM overlay, PiCaSO.

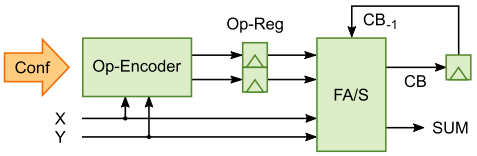

Fig. 4. Architecture of the bit-serial ALU of PiCaSO.

Engine consists of a scalable 2D array of GEMV Tiles, with each Tile containing a local Tile Controller. The size of the 2D array configuration of GEMV tiles is variable, allowing DA-VinCi to scale compute up to an FPGA chip's maximum BRAM density. This brings reuse and portability as the same program can be run on DA-VinCi overlays configured for different FPGA logic families. A unique design aspect of DA-VinCi's architecture is the inclusion of a local controller within each tile. This allows the formation of a hierarchical fanning out of global pipelined control signals. This proved critical for maintaining a clock frequency of the BRAM's Fmax in systems with large numbers of PEs (up to 60K on a xc7vx1140tflg V7 and 86K on a xcvu29p-figd US+ devices). This is discussed in further detail in Section 5.

Each GEMV tile consists of a 2D array of PIM tiles, with each PIM tile consisting of an array of bit-serial PE's matched to a BRAM. The specific organization and implementation of the bit-serial PE's is discussed in Section 4.

4 PIM Tile Architecture

In this section, we discuss the internal architecture of DA-VinCi's PIM tile and its distinct features that help achieve our performance and scalability goals. DA-VinCi's PIM tile is called PiCaSO and is shown in Figure 3. PiCaSO is available at [62] as an open source implementation and is freely available for study, use, modification, and distribution without restriction.

4.1 PiCaSO Datapath

Most PIM architectures use simple datapaths to move data from the BRAM's output ports through bit-serial PEs and loops back to BRAM inputs [26, 29, 63]. This datapath lacks cross-PE reduction, crucial for matrix-vector operations in deep-learning applications. As a workaround, data must either be copied across bit-lines before reduction which adds latency, or include an additional external reduction unit which defeats the purpose of having a PIM architecture. PiCaSO overcomes these limitations by redesigning the datapaths as shown in Figure 3. An **Operand-Multiplexer (Op-Mux)** module between the BRAM and ALU multiplexes bit lines, enabling zero-copy PIM block-level reductions. The network module (Network Node) allows data movement to overlap with computation by streaming bit-serial operands across the network to the ALU of a remote PIM block for reduction operations.

The datapaths are then pipelined to match latencies to the BRAM Fmax clock frequency. Several alternatives were explored to determine where to place registers to pipeline the datapath. Registers were placed and evaluated at BRAM output ports, the output of the Op-Mux module, and the ALU. Figure 3 shows the pipeline configuration that achieved the BRAM Fmax. This configuration requires four-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 four-clock cycle latency overhead was then hidden by designing a controller **Finite State Machine (FSM)** that transitions

Table 1. Bit-Serial ALU (FA/S) Op-Codes

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

FA, Full-Adder.

Table 2. Op-Encoder Configurations for Booth's Radix-2

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

through 4×Read−4×Write states. As a result, only 2N clock cycles are required to process N-bit operands.

4.2 PiCaSO Register File

Similar to traditional PIM architectures [26, 29, 63] PiCaSO uses a BRAM column to serve as the register file for a bit-serial PE. One important design choice for a BRAM-based PIM tile is to determine the optimal organization of the PE array. A square array may at first seem reasonable: a PIM tile with 16 PEs can be logically organized into a 4×4 array [26]. However, to get the best placement and routing results per FPGA logic family, the logical structure of a PIM tile 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 BRAMs. To map an array processor of square dimensions like 256×256 in those devices, a wider PE array per tile (no. of columns $\approx 10 \times$ no. of rows) will have a more regular placement result than a square PE array. For this reason, we designed the PIM tile with a logical array dimension of 1×16 for Virtex FPGAs. We refer to this logical organization of the PIM tile as a PIM Block. This wide aspect ratio of the PE array reduced the number of PIM blocks per row at the system level, thus reducing the number of hops and end-to-end latency when moving data through the reduction network for row-wise accumulation.

4.3 PiCaSO Bit-Serial ALU

In deep-learning applications, the most frequent operation is MAC followed by the addition of the bias vector. Accumulation requires data movement across multiple PIM blocks, managed by PiCaSO's data network and Op-Mux module, while the ALU handles multiplication and addition. In a bit-serial PE, multiplication is typically implemented through repeated 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. The four ALU operations shown in Table 1 support these fundamental tasks. ADD and SUB serve as standard arithmetic functions and are used to update the partial product in multiplication. CPX and CPY enable min/max pooling or other filter operations that require selecting between operands, with CPX also acting as the NOP step in Booth's algorithm. As shown in Figure 4, the bit-serial ALU architecture has two main parts: the **Full-ADD/SUB (FA/S)** module and the **Op-Code Encoder (Op-Encoder)**. FA/S implements the four operations in Table 1 using the following logic functions:

$$\text{SUM} = \text{X} \oplus \text{Y} \oplus \text{CB}_{-1}, \text{ if ADD or SUB} \qquad \text{CB} = \begin{cases} \text{X} \cdot \text{Y} + \text{X} \cdot \text{CB}_{-1} + \text{Y} \cdot \text{CB}_{-1}, & \text{if ADD} \\ \overline{\text{X}} \cdot \text{Y} + \overline{\text{X}} \cdot \text{CB}_{-1} + \text{Y} \cdot \text{CB}_{-1}, & \text{if SUB} \end{cases}.$$

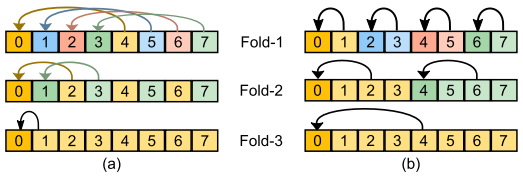

Fig. 5. Folding patterns in Op-Mux.

Table 3. Op-Mux Configurations

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

[a] A[HQ1]: first half-quarter of A.

Op-Encoder provides an abstract interface to the FA/S module by receiving a configuration code, translating it to the correct ALU Op-Code, and storing it in the Op-Reg register. In PiCaSO, this enables Booth's Radix-2 multiplication, where a 3-bit configuration code (Conf) maps to the ALU Op-Code as shown in Table 2. If Conf's MSB is 1, the Op-Code depends on the YX bits, which provide multiplicand bits for each Booth's Radix-2 iteration. The Op-Encoder can be adapted for other operations, like pooling, without modifying the FA/S module.

4.4 PiCaSO Op-Mux

The Op-Mux module enables zero-copy reduction between the PEs within a PIM block and provides a datapath for the overlapping computation and data movement during reductions across multiple PIM blocks. In traditional NEWS networks, reduction works by moving data from a PE register to a register of its neighboring PE [24–26, 28, 29]. In this approach, the q PEs in a row require $q-1$ 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 takes 2 cycles per bit resulting in $2N$ cycles per ADD operation. Performing a reduction between q PEs requires at least $\log_2 q$ steps. Thus, we can represent the **PIM Block-level Reduction Latency (PB-RL)** using the following equation:

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

The Op-Mux module makes use of the folding patterns shown in Figure 5 to perform the reduction. In pattern (a), after adding an operand with its fold-1 pattern, PE 0, 1, 2, and 3 contain the sum 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 sum 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 result will be stored in PE-0. The PB-RL using this folding technique is given by Equation (2), which eliminates the linear term from Equation (1), making it significantly faster:

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

In PiCaSO, we select seven Op-Mux configurations shown in Table 3 matching the fold patterns in Figure 5(a). The default A-op-B configuration keeps Op-Mux transparent by connecting A to X and B to Y. The output port Y is used to implement the folding. For fold configurations, both X and Y receive combinations of port A bits, allowing port B to simultaneously write the ALU output to the destination register, further reducing cycle latency in Equation (2). With four pipeline stages in the datapath, the equation for the reduction latency using the Op-Mux module becomes:

$$\text{PB-RL}_{\text{opmux}} = (N + 4) \cdot \log_2 q. \tag{3}$$

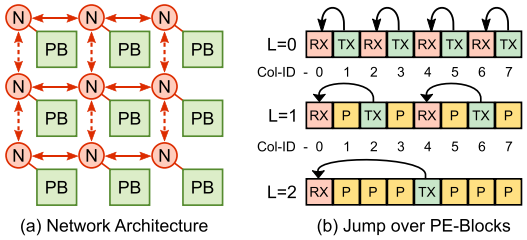

Fig. 6. Binary-hopping data network for fast reduction operations.

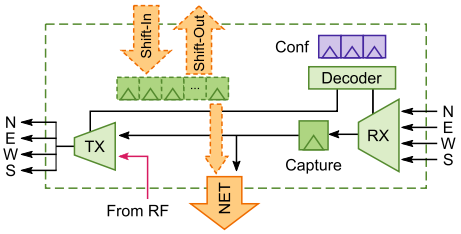

Fig. 7. Architecture of the network node for binary hopping.

In a PIM block with $q = 16$ and $N = 32$, PB-RL$_{\text{NEWS}}$ takes 736 cycles, while PB-RL$_{\text{opmux}}$ takes 144 cycles to perform a block level accumulation, providing a 5.1× improvement over the NEWS network.

4.5 PiCaSO Data Network Architecture

PiCaSO's network architecture uses a PIM block instead of a PE as a node, unlike a standard NEWS network design [26]. As shown in Figure 6(a), a PB connects to the network module, which in turn connects to the network. Block-level accumulation stores results in each block's PE-0 register, so only these registers need accumulation to obtain the row-wide result. Figure 6(b) illustrates the data movement for row-wise reduction. Each network module functions as a flip-flop with routing logic, implementing a conceptual binary tree using a hopping mechanism. Each node has unique row and column IDs corresponding to its position in the array. A node can be configured in one of three modes—Transmitter (TX), Receiver (RX), and Pass-through (P)—based on a Level value (L) and its unique ID.

As shown in Figure 6(b), level 0 (L = 0) connects adjacent nodes by assigning even columns as receivers and odd columns as transmitters, with a network latency of one hop. At level 1, in every group of four nodes, the first acts as a receiver, the second and fourth as pass-through, and the third as a transmitter. This effectively connects the first and the third nodes, with a network latency of two hops. Level 2 connects node-4 to node-0 with a network latency of four hops. In general, the number of hops (H) needed for a given level can be represented by $H = 2^{\text{L}}$. As each hop takes 1 cycle, H also represents the network latency in cycles. At the end of level 2, PE 0 of node-0 contains the accumulation result of the entire row of the array. Each level of the binary tree effectively allows a jump over intermediate PIM blocks. In a network with D nodes in a row, the number of such jumps (J) needed can be expressed as $J = \log_2 D$.

4.6 PiCaSO Network Node Architecture

Figure 7 shows the architecture of the network module that acts as a Node (N) in the data network. It has three logical parts: receiver multiplexer (RX), transmitter multiplexer (TX), and capture registers. It also has a configuration register that holds the current level value (L) 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. Accumulation operation along the columns is not essential in most deep-learning applications. However, for convolution and pooling operations in CNNs, each PE needs to access its adjacent PEs. This can be enabled using a set of flip-flops acting as a **Shift Register (SREG)** along PIM block columns. In a PIM block of dimension 1× 16, we added 15 optional flip-flops for column-wise shifting alongside the capture register at the RX mux output.

4.7 PiCaSO Accumulation Latency

Suppose, we have a $q \times q$ PE array, operating on N-bit operands, with a PIM block of 1×16 PEs connected to the network shown in Figure 6 through the node shown in Figure 7. The block-level accumulation latency from Equation (3) is given by:

$$\text{PB-RL}_{\text{opmux'}} = (N + 4) \cdot \log_2 16 = 4N + 16. \tag{4}$$

Using the terms H and J defined in Section 4.5, we can compute the total hop latency for accumulation. Number of jumps needed for q PEs is $J = \log_2(q/16)$, from level 0 to level $J - 1$, in order. Thus, the total cycle latency to perform all jumps over q PEs is given by Equation (5). The last part of the accumulation is to stream the output of the network capture register through the ALU and ADD it to an operand stream in the receiver PIM 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 (WB)** latency for all jumps is given by Equation (6):

$$H_q = \sum_{L=0}^{J-1} 2^L = 2^J - 1 = 2^{\log_2(q/16)} - 1 == \frac{q}{16} - 1 \tag{5}$$

$$\text{WB}_q = (N + 4) \cdot J. \tag{6}$$

From Equations (4)–(6) we compute the total reduction latency at the **Array Level (Arr-RL)** as follows:

$$\begin{aligned} \text{Arr-RL} = \text{PB-RL}_{\text{opmux'}} + H_q + \text{WB}_q &= 4N + 16 + \frac{q}{16} - 1 + (N + 4) \cdot J \\ &= 15 + \frac{q}{16} + 4N + (N + 4) \cdot J. \end{aligned} \tag{7}$$

As observed in Equation (7), 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.

5 GEMV Engine

The GEMV engine illustrated in Figure 8(a) was initially developed as a standalone accelerator. The standalone version called IMAGine [36] is open source and freely available for study, use, modification, and distribution without restriction at [35]. The GEMV engine 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 SREGs to read out the final result. The input registers receive instructions for 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 SREGs. At the end of the GEMV operation, the output vector is stored in the column SREGs, which can be shifted up and read through the FIFO-out port, one element per cycle.

5.1 GEMV Tile

The GEMV tile shown in Figure 8(b) is the basic building block in the GEMV engine. It consists of (1) an FSM-based controller, (2) a 2D array of PIM blocks, and (3) a fanout tree between them. The controller receives the instruction from the input registers at the top level in Figure 8(a), decodes and generates a sequence of control signals to execute the instruction. The fanout tree distributes 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

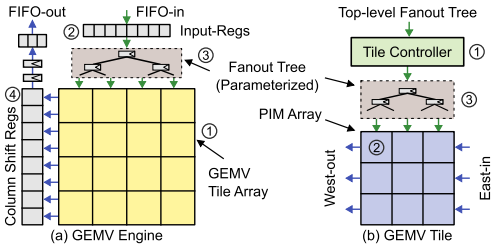

Fig. 8. System architecture of IMAGine: data and instruction flow (a) through the GEMV engine and (b) within a GEMV tile.

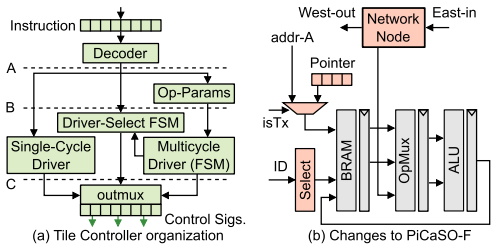

Fig. 9. Design of (a) the GEMV controller and (b) additional control logic in the PIM tile for the GEMV algorithm.

on each side. During accumulation, partial results move from east to west through the PIM arrays, ultimately accumulating in the left-most PE column of the leftmost GEMV tiles in the array.

5.2 GEMV Tile Controller

The tile controller shown in Figure 9(a), is optimized for high clock frequency by minimizing logic depth. Since precise logic depth estimation is difficult during RTL design, the combinatorial logic is partitioned into distinct steps with optional pipeline stages indicated by the dashed lines A, B, and C in Figure 9(a). The controller processes 30-bit instructions, which are handled by either a single-cycle or multicycle driver selected by a two-state FSM based on the opcode. The single-cycle driver completes an instruction each cycle, while the multicycle driver handles more complex operations like ADD, SUB, and MULT, requiring multiple cycles. All inputs and outputs are registered at the module boundary to confine timing paths within the controller.

5.3 PIM Array

Figure 9(b) shows the additional control logic added to the PIM block to build the PIM array within a tile and execute GEMV instructions efficiently. For the GEMV operation, data movement along the North–South direction of the array was not essential. Thus, the network module was simplified keeping only the logic needed for East-to-West data movement. Control signals for selectively enabling/disabling a block based on a given block ID were included. The accumulation algorithm needed three addresses to maximize the overlap of data movement and computation. An additional pointer register was added to each PIM tile to support the third address.

The PIM array is implemented as a parameterized module that instantiates and connects PIM blocks to build a 2D array. The East-in and West-out ports of the network modules of the rightmost and the leftmost columns are directly connected to the East-in and West-out ports of the GEMV tile, respectively. As a result, a 2D array of GEMV tiles creates a 2D array of PIM tiles, where a group of PIM tiles share a controller.

6 VV-Engine Architecture

In deep-learning applications, GEMV/GEMM operations are followed by element-wise vector operations and non-linear activations, which provide substantial parallelism. Non-linear functions like Sigmoid and Tanh, which may take 10–100 CPU cycles, can execute in only a few cycles on a custom ALU. DA-VinCi includes a VV-Engine to accelerate these VV operations. As shown in Figure 10(b), the VV-Engine connects to the GEMV Engine via a serial-SREG interface and consists of vector tiles with vertical data movement only.

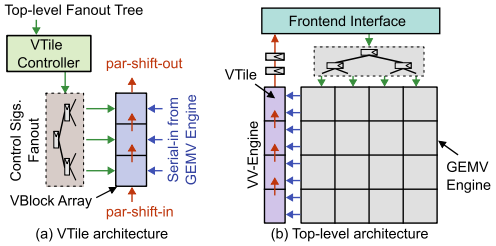

Fig. 10. Architecture of (a) vector tile (b) DA-VinCi system with compute engines connected to the front-end interface.

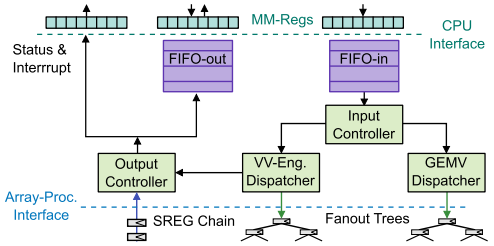

Fig. 11. Design of DA-VinCi front-end interface.

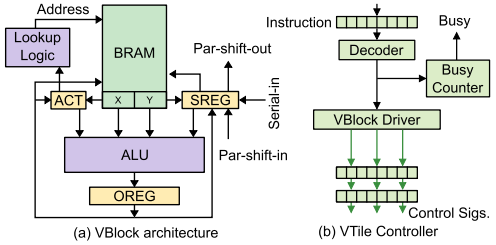

Fig. 12. The architecture of VTile components: (a) VBlock architecture and (b) VTile controller.

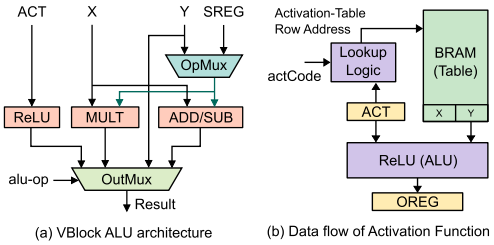

Fig. 13. VBlock computing architecture: (a) ALU architecture and (b) dataflow of lookup-based activation functions.

6.1 Vector Tile

Figure 10(a) illustrates the architecture of a **VV-Engine Tile (VTile)**, which is designed to process 1D vectors and bears a close resemblance to the GEMV tile. Each VTile features a tile controller that connects to an array of **VV Processing Blocks (VBlocks)** via a fanout tree of control signals. The output vectors of GEMV operations are copied to the VTiles through a serial interface. At the end of processing, the result vector is shifted upward to the front-end interface to be pushed into an output FIFO.

6.2 VBlock

Figure 12(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 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 LUTs for several non-linear activation functions. After the GEMV engine processes the input vector, the resultant vector is copied to the SREGs. This vector can then be used as an operand for the ALU or can be stored in the register file. 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. This chain runs through the entire VV-Engine, which is used to shift out the result vector to the front-end interface.

6.3 VTile Controller

The VTile controller architecture, shown in Figure 12(b), is a simplified version of the GEMV tile controller. It includes a VBlock driver module that functions like the single-cycle driver in the

GEMV controller. However, unlike the GEMV controller, it does not have a multicycle FSM driver. Instead, the VTile controller manages multicycle instructions by holding control signals for the required number of cycles. This is done by the Busy-Counter module, which maintains control signals until the instruction completes. This architecture allows for efficient handling of vector operations within the VTile, maintaining high performance while simplifying the control logic.

6.4 VBlock ALU

The VBlock ALU architecture, shown in Figure 13(a), consists of an adder/subtractor, a fixed-point multiplier using a DSP block, and a combinatorial block for the ReLU activation function. The ALU takes four inputs: the **Activation Function Input (ACT)**, port X and port Y from the BRAM, and the SREG. Based on the ALU opcode, a subset of these inputs is selected to perform the required computation. All results pass through an output multiplexer (OutMux), which selects one result as the ALU output. Additionally, the Y input is directly connected to the OutMux, enabling data movement within the register file through OREG.

Figure 13(b) shows the data flows of activation functions. 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 and resource-efficient compared to the fixed-function architectures of activation functions [25–27]. During the initialization of DA-VinCi, BRAMs are preloaded with LUTs of all the non-linear activation functions needed for the target applications. To compute the activation value of a fixed-point input, the input 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 corresponding to the fixed-point value in the ACT register. The output is then stored in the OREG. This entire lookup process is completed in three cycles.

7 DA-VinCi Control and Programming Interface

Figure 11 illustrates the system-level architecture of DA-VinCi. 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. After the GEMV operations, 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 the SREG-chain in the VV-Engine.

7.1 Front-End Interface Architecture

Figure 11 illustrates DA-VinCi's front-end interface architecture. This interface comprises memory-mapped registers, two FIFOs, status registers, and various controller blocks. The memory-mapped registers allow the front-end CPU to write data and control words to the FIFOs and read accelerator outputs and status. The FIFO-in receives 32-bit instruction words, while the FIFO-out holds the latest result vectors from the VV-Engine. When the last vector element is written to FIFO-out, an interrupt signal notifies the CPU that the result is ready for reading.

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-Engines 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 4. ISA of DA-VinCi

Instruction		Operands	Semantics
GEMV Subset	**VV Subset**		
mv_write	vv_write	addr, data	BRAM[addr] <= data[a]
mv_nop	vv_nop	-	Spend 1 cycle doing nothing
mv_mov	vv_mov	rd, rs	RF[rd] <= RF[rs][a]
mv_add	vv_add	rd, rs1, rs2	RF[rd] <= RF[rs1] + RF[rs2] (rd = OREG for vv_add)
mv_sub	vv_sub	rd, rs1, rs2	RF[rd] <= RF[rs1] − RF[rs2] (rd = OREG for vv_sub)
-	vv_mult	rs1, rs2	OREG <= RF[rs1] × RF[rs2][b]
-	vv_activ	actCode	OREG <= activation-func<actCode>(ACT)[c]
mv_selectBlk	-	rowID, colID	Select all BRAMs in the GEMV[rowID] for mv_write
mv_selectRow	vv_selectBlk	rowID	Select all BRAMs in the row = rowID for mv/vv_write (only one Vblock per row in VV-Engine)
mv_selectCol		colID	Select all BRAMs in the GEMV[:, colID] for mv_write
mv_selectAll	vv_selectAll	-	Select all BRAMs in the GEMV/VV-Engine for mv/vv_write
mv_accumRow	-	level, reg	GEMV row-wise accumulation of RF[reg] using L=level
mv_updatepp	-	ppreg, mpcand, mplier, bitNo	RF[ppreg] += RF[mpcand] × RF[mplier][bitNo]
-	vv_shiftOff	-	Disable all shifting of SREG
-	vv_serialEn	-	Enable serial shifting of SREG
-	vv_parallelEn	-	Enable parallel shifting of SREG chain

[a]BRAM[addr] semantics access BRAMs directly using row address. RF[reg] semantics access BRAMs as bit-serial registerfiles for the GEMV subset and as vector registerfiles for the VV subset.
[b]Fixed-point multiplication.
[c]Activation-func<actCode> selects between ReLU or non-linear LUTs.

The output controller monitors the shifted-out vectors through the SREG chain of the 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.

7.2 DA-VinCi Instruction Set

DA-VinCi has its own ISA, defined independently from the hardware of submodules like the GEMV and VV-Engines. The front-end interface translates 32-bit instruction words into the format required by each module, making the ISA and compiled programs portable across future generations of DA-VinCi.

7.2.1 GEMV Engine ISA Subset. The first column of Table 4 lists the ISA subset for controlling the GEMV engine. These assembly instructions are prefixed by "mv_" to indicate matrix-vector operations. These instructions allow selective BRAM writes in the GEMV array, data movement within the register file, arithmetic operations like ADD and SUB, and row-wise accumulation. A key instruction is the update-partial-product (updatepp) instruction, which executes one step of Booth's multiplication algorithm. It updates the partial-product register (ppreg) by adding it to the product of the multiplicand register (mpcand) and the multiplier bit (mbit), performing the necessary operand shifts. Repeated iterations of this instruction with successive mbit values complete the multiplication between the mpcand and mplier registers in the GEMV engine.

7.2.2 VV-Engine ISA Subset. The second column of Table 4 lists the ISA subset for controlling 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 selective BRAM writes in the VBlock array, data movement between registers, and essential operations like ADD, SUB, MULT, and activations. The "vv_active" instruction is unique to the VV-Engine, enabling selection of non-linear activation functions from BRAM or the ReLU function via the 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 activation LUTs, provides the flexibility needed for a wide range of activation functions used in deep-learning models.

The VV-Engine subset also includes instructions for controlling the shift-register chain within the VV-Engine. SREG serial shifting can be enabled to copy vectors from the GEMV engine, disabled to retain data during computation, or set to parallel shifting to output the result vector to FIFO-out. These instructions provide complete control over the dataflow within the VV-Engine essential for maintaining high efficiency and performance.

7.3 Assembler Design

In designing the software toolchain for DA-VinCi, a long-term vision would ideally involve the integration of a mature compiler framework such as LLVM [64] or MLIR [65], which could provide advanced optimizations, dialect extensions, and broader portability. However, for the purpose of rapid prototyping and experimentation, we adopted a more lightweight and flexible approach. We implemented the DA-VinCi assembler as a standalone Python module, prioritizing ease of use, simplicity, and extensibility. This design decision enabled rapid development cycles and integration with existing Python-based toolchains commonly used in deep-learning research. Furthermore, Python's rich ecosystem allows the assembler to be easily extended with powerful numerical libraries such as NumPy, enabling efficient preprocessing of input vectors, parameter quantization, or even automatic instruction stream generation for common network layers. This Python-based assembler serves as a practical and effective solution for prototyping, while also providing a foundation that can be incrementally transitioned to a more formal compilation infrastructure in the future. The Python-based DA-VinCi assembler is open source and freely available at [66] for study, use, modification, and distribution without restriction.

The assembler design adheres to a structured and modular philosophy. Each low-level instruction from the DA-VinCi ISA presented in Table 4 is exported as a dedicated Python function. This one-to-one mapping allows for clear abstraction and reuse, enabling users to write high-level scripts that directly invoke ISA primitives. To further enhance usability and reduce verbosity, the assembler also includes a set of macros tailored for common operations in deep-learning workloads. These macros are constructed on top of the low-level ISA functions as well as PiCaSO's low-level instructions, offering composable units for higher-level tasks such as fixed-point multiplication and reduction.

Listing 1 presents two such macros: `mv_MULTFXP` and `mv_ALLACCUM`. The `mv_MULTFXP` macro performs a fixed-point multiplication using Booth's Radix-2 algorithm by repeatedly issuing `mv_updatepp` instructions across all bits of the multiplier, followed by a register shift to adjust for the fractional part. The `mv_ALLACCUM` macro performs a global array accumulation by first invoking `mv_BLOCKACCUM` to perform local reductions within each block and then sequentially applying `mv_accumRow` across increasing levels of the binary-hopping network.

Listing 2 shows the `mv_BLOCKACCUM` macro, which wraps multiple intra-block fold operations using PiCaSO's `accumblk` instruction. The `mv_SYNC` macro issues two consecutive `nop` instructions as a synchronization barrier for the GEMV Engine. Similarly, the `vv_SYNC` macro issues one `nop` instruction as a synchronization barrier for the VV-Engine. These `nop` instructions fill up the

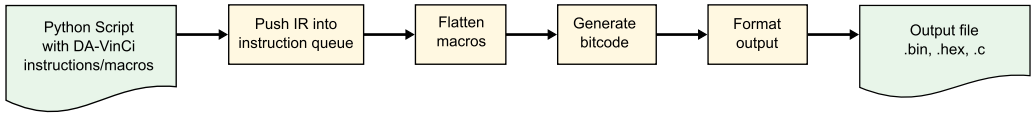

Fig. 14. Compilation flow of the DA-VinCi assembler from high-level Python script to final output bitcode.

Listing 1. Fixed-point multiplication and array accumulation macros

```
def mv_MULTFXP(rd, multiplicand, multiplier):
    for bitNo in range(regWidth):
        mv_updatepp(resvReg0, multiplicand, multiplier, bitNo)
    picaso.movOffset(offset=fracWidth, ppreg, resvReg0)

def mv_ALLACCUM(rd, rs):
    mv_BLOCKACCUM(rd=rd, rs=rs)
    for l in range(0, maxLevel+1):
        mv_accumRow(level=l, reg=rd)
```

Listing 2. block accumulation and sync. macros

```
def mv_BLOCKACCUM(rd, rs):
    for f in range(1, maxFold+1):
        picaso.accumblk(fold=f, rd=rd, rs=rs)

def mv_SYNC():
    mv_nop(); mv_nop()

def mv_SYNC():
    vv_nop()
```

instruction pipeline just enough to stall the dispatcher modules from issuing new instructions until the old ones have been executed.

It is important to note that the implementations shown in Listings 1 and 2 are simplified versions of the actual assembler macros. They are intended to illustrate the core logic and structural flow of the macros, removing implementation details such as manipulation of the internal data structures of the assembler. The complete and fully functional implementations, along with many other macros useful for deep-learning workloads, are available in the open source code repository [66] for further study and extension.

The end-to-end transformation of a user-defined DA-VinCi program into executable bitcode is illustrated in Figure 14. The process begins with a Python script that invokes a sequence of instructions and macros using the assembler API. Each of these function calls appends a structured **Internal Representation (IR)**, implemented as a Python dictionary, to an instruction queue, which is a Python list. In the next stage, high-level macros are flattened into several machine instructions from the ISA.

The machine code generation step then encodes these instructions into 32-bit binary representations, producing architecture-compatible bitcode. This bitcode is subsequently passed to a formatter that generates the final output in one of several supported formats. Specifically, the assembler supports a binary text format compatible with Verilog's $readmemb() function, allowing seamless integration with hardware simulation environments. Additionally, it supports a uint 32-formatted C array for importing the bitcode into system-level C/C++ projects. The formatter can optionally embed inline comments into the exported files to enhance traceability and debugging, making it easier to correlate high-level macros with their compiled binary equivalents. This transformation pipeline enables rapid deployment of DA-VinCi programs in both simulation and real hardware environments.

7.4 Decomposing Deep-Learning Models into DA-VinCi's ISA

To illustrate the process of mapping deep-learning workloads to DA-VinCi's instruction set, we focus on two widely used models: a single-layer MLP and one iteration through a LSTM cell. While these two examples serve as representative case studies, it is important to note that any model structured around generalized matrix-vector (GEMV/GEMM) operations followed by vector point-wise transformations, such as non-linear activations or element-wise arithmetic, can be decomposed using the same methodology.

The mathematical formulations for both MLP and LSTM computations are well established in the literature and can be found in prior publications such as [67, 68] for MLPs and [69–71] for LSTM cells. The mathematical formulation of these two models is shown in Equations (8)–(15). For the MLP case, a single inference step consists of a linear transformation shown by Equation (8), followed by a non-linear activation as shown by Equation (9). where W, b, and x_t represent the weight matrix, bias vector, and input vector, respectively. This computation maps directly to a GEMV operation followed by a vector ReLU, both of which are natively supported by DA-VinCi's ISA:

$$z_t = Wx_t + b, \tag{8}$$

$$h_t = \mathrm{ReLU}(z_t). \tag{9}$$

For the LSTM cell, the computation is more involved and consists of six Equations (10)–(15). It begins with gate computations—input i_t, forget f_t, and output o_t—using affine transformations followed by a sigmoid activation Equations (10)–(12). A candidate cell state $\tilde{c}_t$ is computed using a tanh activation (Equation (13)). The final cell state c_t and hidden state h_t are updated via gated combinations of current and previous states (Equations (14) and (15)). Each of these steps can be expressed as a combination of GEMV and vector-level operations, and thus can be realized using sequences of DA-VinCi instructions and macros:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \tag{10}$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f), \tag{11}$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o), \tag{12}$$

$$\tilde{c}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_f) \tag{13}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{14}$$

$$h_t = o_t \odot \tanh(c_t). \tag{15}$$

Listing 3 demonstrates how a single-layer MLP inference, as defined by Equations (8) and (9), is mapped onto DA-VinCi's instruction set using the Python-based assembler. The script begins by importing the assembler module and loading system parameters from a YAML configuration file. It then allocates registers for use in both the GEMV Engine and VV-Engine operations. Registers are allocated for the input vector, weight matrix, bias vector, and temporary registers to hold intermediate products and accumulations. The GEMV computation is initiated using the `mv_MULTFXP` macro, which multiplies the input vector with the weight matrix and stores the result in a product register. This result is then accumulated using `mv_ALLACCUM` to compute the dot products across rows. After issuing a `mv_SYNC` instruction to ensure the GEMV Engine has completed processing, control shifts to the VV-Engine for the point-wise operations.

Next, the program disables the SREG, adds the bias vector using `vv_add`, and stores the result in the OREG, which represents the linear transformation $z_t = Wx_t + b$. It then moves the output into the activation register and applies the ReLU function using `vv_activ`, corresponding to $h_t = \mathrm{ReLU}(z_t)$. The final result is moved back into the SREG for output, and parallel shifting is enabled to stream the results into the output FIFO. Lastly, the compiled bitcode is exported in two formats: a Verilog-compatible binary for simulation and a C-file for system-level integration.

Listing 4 shows a part of the implementation of a single LSTM cell iteration based on Equations (10)–(15), illustrating how the core operations of the LSTM cell are expressed using DA-VinCi's instruction set. The complete script follows the same general structure as the MLP example and requires the same import statement from the assembler module as well as the export functions for generating the final bitcode, which are not shown in the listing for brevity. Before computation, registers are first allocated for all LSTM weight matrices, bias vectors, and input vectors. The four nonlinear components of the LSTM—input gate i_t, forget gate f_t, output gate o_t, and candidate cell state $\tilde{c}_t$—are each computed by invoking the `computeGate()` function, which encapsulates GEMV computation and activation.

The cell state update, $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$, is implemented using a sequence of `vv_mult`, `vv_mov`, and `vv_add` operations that perform point-wise multiplication and addition on the intermediate results. The new cell state is stored in both `vregCt` and `vregCp`, the latter serving as c_{t-1} for the

Listing 3. Complete Python script for MLP kernel generation

```
from davinci_assembler import *

# Assembler parameters setup
davinci_as.loadParams('davinci_params.yml')

# Register allocation
# VV-Engine registers
vregB = 1     # bias vector
# MV-Engine registers
mregM = 0     # weight matrix
mregV = 1     # input row-vector
mregProd = 4  # for storing intermediate product
mregAcum = 5  # for storing intermediate accumulation

# -- Perform MV-Engine computations
# SREG = W*X
mv_MULTFXP(rd=mregProd, multiplicand=mregV,
           multiplier=mregM)
vv_serialEn() # enable serial shifting before accumulation
mv_ALLACCUM(rd=mregAcum, rs=mregProd) # Perform accumulation
mv_SYNC()     # Wait till the accumulation finishes

# -- Perform VV-Engine computations
vv_shiftOff()             # disable all shifting
vv_add(vregB, VVREG.S)    # OREG = W*X + B
# Apply activation
vv_mov(VVREG.ACT, VVREG.O)
vv_activ(ACTCODE.RELU)    # OREG = ReLU(W*X + B)
# Shift out the result
vv_mov(VVREG.S, VVREG.O)
vv_parallelEn()           # parallel shift out to FIFO-out

# Export output for simulation and DA-VinCi driver
davinci_as.export_verilogBin('davinci_prog_mlp.bin')
davinci_as.export_CprogHex( 'prog_mlp','davinci_prog_mlp.c')
```

Listing 4. Partial Python script for LSTM kernel generation

```
# Register allocation
mregWxi = 0; mregWxf = 1; mregWxo = 2; mregWxc = 3
mregWhi = 4; mregWhf = 5; mregWho = 6; mregWhc = 7
vregbi = 0; vregbf = 1; vregbo = 2; vregbc = 3
# input registers
mregXt = 20; mregHp = 21; vregCp = 20
... [truncated]

# compute It, Ft, Ot, and ~Ct (C_)
computeGate(vregIt, ACTCODE.SIGM, mregIa,
            mregWxi, mregWhi, vregbi)
computeGate(vregFt, ACTCODE.SIGM, mregFa,
            mregWxf, mregWhf, vregbf)
computeGate(vregOt, ACTCODE.SIGM, mregOa,
            mregWxo, mregWho, vregbo)
computeGate(vregC_, ACTCODE.TANH, mregCa,
            mregWxc, mregWhc, vregbc)

# Ct = Ft . Cp + It . C_
vv_mult(vregFt, vregCp)
vv_mov(vregTmp, VVREG.O) # vregTmp = Ft . Cp
vv_mult(vregIt, vregC_)
vv_mov(VVREG.S, VVREG.O) # SREG = It . C_
vv_add(vregTmp, VVREG.S) # OREG = Ft . Cp + It . C_
vv_mov(vregCt , VVREG.O) # Ct = Ft . Ct-1 + It . Ca
vv_mov(vregCp , VVREG.O) # Cp = Ct (for next iteration)

# Ht = Ot . tanh(Ct), then shift out
vv_mov(VVREG.ACT, vregCt)
vv_activ(ACTCODE.TANH)
vv_mov(VVREG.S, VVREG.O) # SREG = tanh(Ct)
vv_mult(vregOt, VVREG.S) # OREG = Ht = Ot . tanh(Ct)
vv_mov(VVREG.S, VVREG.O) # SREG = Ht
vv_parallelEn()          # parallel shift out to FIFO-out
vv_SYNC()
```

next iteration. Finally, the hidden state $h_t = o_t \odot \tanh(c_t)$ is computed by applying a tanh activation to the cell state, multiplying it with the output gate vector. The resultant vector is shifted out using the VV-Engine's parallel SREG chain. A `vv_SYNC` is issued at the end to ensure that all operations complete before any new instruction can be issued.

Listing 5 provides the implementation of the `computeGate` function used in Listing 4. This function abstracts the common GEMV-and-activation pattern used across all four gate computations. It first performs a matrix-vector multiplication between the current input x_t and the corresponding weight matrix W_x, followed by a reduction using `mv_ALLACCUM`. A similar multiplication and accumulation is then performed for the hidden state from last iteration h_{t-1} with the weight matrix W_h. The two results are added together using `mv_add` and stored in a destination matrix register. Note that the serial shifting for the VV-Engine is enabled using `vv_serialEn` right before `mv_add`. As a result, the resultant vector is simultaneously copied to the SREG special register of the VV-Engine. The GEMV synchronization barrier, `mv_SYNC`, ensures that the `mv_add` instruction is completed before the `vv_shiftOff` instruction is executed on the VV-Engine. Next, the bias vector is added to this resultant vector using `vv_add`, followed by the application of the specified non-linear activation function, either sigmoid or tanh, using `vv_activ`. The final activation output is stored in the destination vector register `vregDest`. Although `computeGate` is defined as a Python function in this script, each invocation will be flattened by the assembler during compilation. As a result, it effectively behaves like a macro in the context of the davinci_assembler module.

Note that these examples present straightforward implementations intended to illustrate the fundamental process of decomposing deep-learning models into DA-VinCi's instruction set. In practice, real-world kernels often require additional considerations, such as partitioning large weight

Listing 5. Implementation of the computeGate() function in the LSTM kernel

```
def computeGate(vregDest, actCode, mregDest, mregWx, mregWh, vregb):
    # compute Wx*Xt
    mv_MULTFXP(rd=mregProd, multiplicand=mregXt, multiplier=mregWx)
    mv_ALLACCUM(rd=mregAcumX, rs=mregProd)
    # compute Wh*Hp
    mv_MULTFXP(rd=mregProd, multiplicand=mregHp, multiplier=mregWh)
    mv_ALLACCUM(rd=mregAcumH, rs=mregProd)
    # compute Wx*Xt + W*Hp, and copy to VVREG.SREG
    vv_serialEn()               # enable serial-shifting to copy result to VVREG.SREG
    mv_add(rd=mregDest, rs1=mregAcumX, rs2=mregAcumH)
    mv_SYNC()                   # Wait till the accumulation finishes
    vv_shiftOff()               # disable serial-shifting for VV-Engine computation
    vv_add(vregb, VVREG.S)      # OREG = Wx*X + W*Hp + b
    # apply activation
    vv_mov(VVREG.ACT, VVREG.O)
    vv_activ(actCode)           # OREG = actFunc(Wx*Xt + W*Hp + b)
    vv_mov(vregDest, VVREG.O)   # vregDest = actFunc(Wx*Xt + W*Hp + b)
```

matrices to fit within the physical array dimensions, or concatenating smaller matrices to improve performance, and so on. Nevertheless, the code listings effectively convey the general methodology for mapping high-level deep-learning models into DA-VinCi's ISA. They also demonstrate how the Python-based assembler can be used to implement, compile, and export these kernels for simulation and execution on the hardware overlay.

7.5 Generated Bitcode

Listing 6 shows the machine-level bitcode generated by the `export_verilogBin` function call in the MLP example presented in Listing 3. The syntax is formatted to be compatible with Verilog and SystemVerilog's `$readmemb` system function, allowing the program to be directly loaded into simulation testbenches. Each line in the listing corresponds to a 32-bit machine code, with underscores inserted to visually separate opcode fields and operands, aiding in readability. The assembler also inserts comments to annotate macro expansions and individual instructions, which significantly simplifies the debugging and verification process. At the top of the listing, the comment shows that the macro `MV_MULT` was flattened into 17 individual instructions from the DA-VinCi ISA. The following lines show individual machine code, each generated by a corresponding API call within the source-level Python script. Note that the assembler uses positive integers to denote general-purpose registers and negative values for special-purpose registers such as SREG, OREG, and ACT. This convention is reflected in the bitcode comments.

Listing 7 presents the same compiled bitcode as in Listing 6, but formatted in C syntax for inclusion in system-level software projects. This version is generated by the `export_CprogHex` function call from the MLP script. This version encapsulates the instruction stream as an array of 32-bit unsigned integers (`uint32_t`), written in hexadecimal representation. Each entry in the array corresponds directly to a DA-VinCi instruction, with one-to-one correspondence in both content and order to the binary representation shown in Listing 6. The accompanying comments, also generated by the assembler, mirror those from the Verilog-compatible output and help maintain traceability between the macro calls and their flattened instruction sequences. This format facilitates seamless integration of the compiled bitcode into C/C++-based runtime environments or embedded firmware targeting DA-VinCi-based systems.

7.6 Program Execution on the Co-Processor

In this subsection, we demonstrate how to execute a compiled DA-VinCi kernel, such as the one shown in Listing 7, on a system where DA-VinCi functions as a co-processor. As illustrated in Figure 2, the system comprises an on-chip processor (e.g., MicroBlaze, ARM, or RISC-V) interfaced with the DA-VinCi overlay co-processor via an AXI bus. While the previous subsection focused on

Listing 6. MLP kernel Bitcode in verilog $readmemb() format

```
// ---- MACRO: MV_MULT rd=60, multiplicand=1, multiplier=0; From macro
       call: MV_MULTFPX rd=4, multiplicand=1, multiplier=0;
00_1000_0000000000_0000000000000000
00_0011_0000111100_0000000001000000
... [13 lines truncated]
00_0011_1110111100_0000000001000000
00_0011_1111111100_0000000001000000
// ---- End of MACRO
00_0111_0000001000_0000000100111100 // MV_MOV_OFFSET offset=8, dest=4,
       src=60
01_01010_0000000000_0000000000000000 // VV_SERIAL_EN
// ---- MACRO: MV_BLOCK_ACCUM rd=5, rs=4; From macro call: MV_ALLACCUM
       rd=5, rs=4;
00_0100_0000000001_0000000101000100
00_0100_0000000010_0000000101000101
00_0100_0000000011_0000000101000101
00_0100_0000000100_0000000101000101
// ---- End of MACRO
00_0100_0001000000_0000000000000101 // MV_ACCUM_ROW level=0, reg=5
00_0100_0001000001_0000000000000101 // MV_ACCUM_ROW level=1, reg=5
// ---- MACRO: MV_SYNC
00_0000_0000000000_0000000000000000
00_0000_0000000000_0000000000000000
// ---- End of MACRO
01_01001_0000000000_0000000000000000 // VV_DISABLE_SHIFT
01_00100_0000000000_0000000100000000 // VV_ADD opl=1, opr=-1 (SREG:-1)
01_10010_0000000000_0000000000000000 // VV_MOV dest=-3, src=-2 (S:-1, O
      :-2, ACT:-3)
01_00111_0000000000_0000000000000000 // VV_ACTIV actCode=0
01_01101_0000000000_0000000000000000 // VV_MOV dest=-1, src=-2 (S:-1, O
      :-2, ACT:-3)
01_01011_0000000000_0000000000000000 // VV_PARALLEL_EN
```

Listing 7. MLP kernel bitcode as C-array for DA-VinCi driver

```
static const uint32_t word_arr[] = {
   // ---- MACRO: MV_MULT rd=60, multiplicand=1, multiplier=0; From
         macro call: MV_MULTFPX rd=4, multiplicand=1, multiplier=0;
   0x20000000,
   0x0C3C0040,
   ... [13 lines truncated]
   0x0FBC0040,
   0x0FFC0040,
   // ---- End of MACRO
   0x1C08013C, // MV_MOV_OFFSET offset=8, dest=4, src=60; From macro
          call: MV_MULTFPX rd=4, multiplicand=1, multiplier=0;
   0x54000000, // VV_SERIAL_EN
   // ---- MACRO: MV_BLOCK_ACCUM rd=5, rs=4; From macro call:
          MV_ALLACCUM rd=5, rs=4;
   0x10010144,
   0x10020145,
   0x10030145,
   0x10040145,
   // ---- End of MACRO
   0x10400005, // MV_ACCUM_ROW level=0, reg=5; From macro call:
          MV_ALLACCUM rd=5, rs=4;
   0x10410005, // MV_ACCUM_ROW level=1, reg=5; From macro call:
          MV_ALLACCUM rd=5, rs=4;
   // ---- MACRO: MV_SYNC
   0x00000000,
   0x00000000,
   // ---- End of MACRO
   0x52000000, // VV_DISABLE_SHIFT
   0x48000100, // VV_ADD opl=1, opr=-1 (SREG:-1)
   0x64000000, // VV_MOV dest=-3, src=-2 (S:-1, O:-2, ACT:-3)
   0x4E000000, // VV_ACTIV actCode=0
   0x5A000000, // VV_MOV dest=-1, src=-2 (S:-1, O:-2, ACT:-3)
   0x56000000}; // VV_PARALLEL_EN
```

writing and compiling programs using the DA-VinCi assembler, our focus here shifts to the design of the driver program that runs on the front-end processor. Figure 11 provides further insight into the internal structure of DA-VinCi's front-end interface, highlighting the role of key components such as the input/output FIFOs, dispatcher units, and control logic.

At a high level, the driver program follows a simple sequence: it initializes DA-VinCi's relevant registers, loads the input data into the required register for a single kernel iteration, loads the compiled kernel bitcode into FIFO-in, waits for the computation to complete, and finally retrieves the output vector. This output can then be consumed or used to trigger the next iteration, depending on the application. Listing 8 shows a minimal C program that runs on the front-end processor for a low-latency inference application, illustrating these steps. The `init_davinci()` function initializes the DA-VinCi system by resetting necessary registers, loading LUTs for activation functions, and transferring model parameters, such as weight matrices and bias vectors, into the appropriate registers expected by the compiled kernel. Inside the main loop, the `readSensor()` function acquires input data, and the `runInference()` function loads the input vector, launches the compiled kernel, waits for its completion, and retrieves the output. The output inference result is then used in the application as desired.

Listing 9 presents a possible implementation of the `runInference()` function for executing the MLP kernel described earlier in Listing 3. All functions prefixed with `dav_` are part of the DA-VinCi driver library, which abstracts the interaction with the front-end interface. The kernel, declared as `prog_mlp`, is wrapped in a `Davinci_Prog` struct in a separate header file. This wrapper struct encapsulates metadata and the compiled instruction array shown in Listing 7, making it convenient to manage the kernel in the application context. The function first loads the floating-point input vector into the matrix register `mregV`, performing fixed-point quantization using the fractional width stored in `prog_mlp`. The `dav_clearEOV()` function clears the **End-of-Vector (EOV)** interrupt flag before execution begins. This interrupt is generated by DA-VinCi every time a result vector is shifted out from the VV-Engine to FIFO-out. The kernel is then loaded into FIFO-in using `dav_pushProgram()`, which starts execution as soon as the first instruction becomes

Listing 8. Driver CPU main() function skeleton

```
int main() {
    init_platform();
    init_davinci(); // clears registers and loads model parameters
    // Free-running application
    float sensData[INPVEC_SIZE];
    while(1) {
        readSensor(sensData);
        int result = runInference(sensData);
        // Do something with the inference result
        xil_printf("Result: %d\n", result);
    }
    cleanup_platform();
    return 0;
}
```

Listing 9. Driver CPU function to run inference on DA-VinCi

```
int runInference(float inpVec[INPVEC_SIZE]) {
    // Load input and run the kernel
    extern Davinci_Prog prog_mlp;
    dav_loadVectorf_row(mregV, inpVec, INPVEC_SIZE, prog_mlp.fracWidth);
    dav_clearEOV();  // clear eovInterrupt flag before kernel execution
    dav_pushProgram(&prog_mlp);
    dav_pollEOV();  // Wait for EOV interrupt

    // Pop the output vector and run argmax() to get the class
    dav_vecval_t vecOut[VECBUF_SIZE];
    dav_popVector(vecOut, VECBUF_SIZE);
    int result = argmax(vecOut, OUTVEC_SIZE);
    return result;
}
```

available in the FIFO. Next, the processor blocks until completion by polling using the function `dav_pollEOV()`. After kernel execution is completed, the output vector is retrieved from FIFO-out using `dav_popVector()`, and a post-processing step such as `argmax()` is applied to obtain the final classification result.

It is worth noting that if the MLP model included multiple layers, each layer would typically be Implemented as a separate kernel in this compilation flow. In such cases, the output from one layer would need to be transferred from the FIFO-out back to FIFO-in before executing the next kernel. In the current simplified implementation, data movements between the input-output FIFOs are handled by the CPU. However, in a performance-critical or real-world deployment, such transfers should ideally be managed by a dedicated DMA engine to reduce CPU overhead. Additionally, the `runInference()` function itself could be restructured as an interrupt-driven routine, triggered by the EOV interrupt generated by DA-VinCi. This would enable greater concurrency and responsiveness, allowing the processor to perform other tasks while the DA-VinCi co-processor is running inference in the background.

8 Evaluation and Results

DA-VinCi is designed to maximize system clock frequency by matching the operating frequency of BRAM and using all available BRAMs as PIM blocks. Achieving this requires every system component to meet these high standards. This section presents a bottom-up analysis of DA-VinCi, starting with the PIM block and extending to system-level performance using real-world deep-learning applications. DA-VinCi was tested on AMD Alveo U55C and VC-707 FPGA boards with Vivado 2022.2.

8.1 Analysis of PIM Overlay

8.1.1 PIM Block Performance and Utilization. To evaluate the proposed PiCaSO PIM architecture, we compare its overlay implementation with the benchmark SPAR-2 overlay from [25–27]. This analysis quantifies the benefits of PiCaSO's design features. Table 5 shows block and tile-level utilization and performance across various pipeline configurations for PiCaSO and SPAR-2 PIM blocks.

The pipeline configuration shown in Figure 3 with all pipeline stages enabled is termed as the Full-Pipe configuration in the table. The Single-Cycle configuration has none of those stages enabled. The RF-Pipe and Op-Pipe configurations only enable the stages after the registerfile and the Op-Mux module, respectively. All these designs were implemented and run on Virtex-7 (xc7vx485) and Alveo U55 FPGAs. Utilization numbers are reported for an array of 256 PEs organized in a 4×4 tile of PIM blocks, with 16 PEs in each block. The total utilization per tile (T) and the average utilization per block (B) are shown. The Full-Pipe configuration achieved a 2.25× and a 1.67× increase in clock

Table 5. Performance and Utilization of Comparison between PIM Blocks (B) and 4 × 4 Tiles (T) of Different Pipeline Configurations

	Benchmark [26]				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	3,023	189	2,449	153	835	52	774	48	895	56	1,068	67	1,017	64	1,064	67	836	52	774	48
FF	1,024	64	768	48	1,799	112	1,799	112	1,031	64	1,031	64	1,543	96	1,527	95	1,543	96	1,543	96
Slice	1,056	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 6. Cycle Latency of Different PIM Operations

Operation	Benchmark [26]	PiCaSO
Add/Sub	$2N$	$2N$
MULT	$2N^2 + 2N$	$2N^2 + 2N$
Accum.	$N(q - 1 + 2\log_2 q)$	$15 + \frac{q}{16} + 4N + (N+4)J$
$q = 128$ N = 32	4,512	259

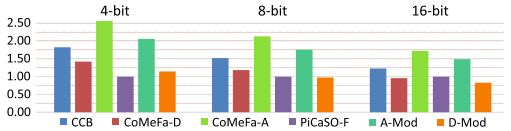

Fig. 15. Relative MAC latency of custom designs w.r.t. PiCaSO.

frequency compared to the benchmark design on Virtex-7 and U55 devices, respectively. In both devices, Full-Pipe provided a 2× improvement in resource (Slice) 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. Op-Pipe achieved better performance compared with RF-Pipe by minimizing the clock latency contributed by the network module.

Table 5 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. Such a high frequency was achievable due to the pipelined architecture, where the slowest stage is the BRAM.

8.1.2 Reduction Network. To study latency improvements in the reduction operation, we compared the clock cycles required for operations supported by PiCaSO and SPAR-2 PIM blocks. Table 6 presents these latencies side-by-side. Both PiCaSO and SPAR-2 use Booth's Radix-2 algorithm for multiplication, resulting in identical ADD/SUB and MULT latencies. SPAR-2 employs a standard NEWS network for operand transfer between PEs during MAC operations. The "Accum." row in Table 6 shows the reduction latency for both architectures, while the last row compares the total latency for accumulating 32-bit operands (N) in a row of 128 PEs (q). PiCaSO's reduction network achieves a 17× latency improvement over SPAR-2, due to the binary-hopping network discussed in Section 4.5, which maximizes data transfer overlap with computation.

8.2 PIM Design Comparison with Custom-BRAM Implementations

Figure 15 shows the MAC latency of existing custom-BRAM PIM architectures relative to the fully pipelined PiCaSO (PiCaSO-F) on Alveo U55 FPGA. Latency is calculated for 16 parallel MULTs followed by accumulation. Clock speeds for the custom-BRAM designs were adjusted based on performance degradations reported in [24, 29]. Except for CoMeFa-D at 16-bit precision, PiCaSO has the lowest latency due to its faster clock and efficient accumulation. CCB and CoMeFa extend the clock period for a full read-modify-write cycle, reducing MULT cycles but increasing latency at

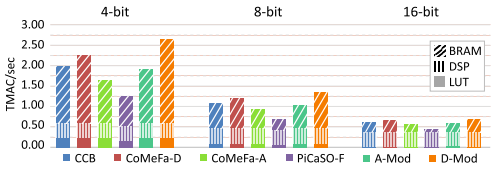

Fig. 16. Peak MAC throughput comparison of PIM designs.

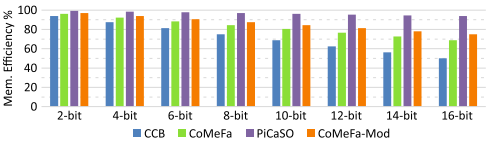

Fig. 17. BRAM memory utilization efficiency on Virtex devices.

Table 7. Utilization and Frequency of 12 × 2 GEMV Tile Components

	Controller	Rel.	Fanout	Rel.	PIM Array	Rel.	Tile
LUT	167	5.8%	0	0.0%	2,736	94.2%	2,903
FF	155	4.0%	615	15.9%	3,096	80.1%	3,866
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

higher precisions. Still, PiCaSO is 1.72×–2.56× faster than CoMeFa-A, the most practical design reported in [29].

Peak TeraMAC/sec throughputs on the U55 FPGA are shown in Figure 16. CCB and CoMeFa design the BRAM IP to support one PE per bitline. With a column muxing factor of 4 [29], 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 CoMeFa designs. This results from PiCaSO not sacrificing the clock speed as in the custom-BRAM designs.

Memory utilization efficiency, an important metric for PIM architectures, is not discussed in [24, 29]. It can be defined as the fraction of BRAM memory available for storing model weights. Both CCB and CoMeFa follow computation techniques from [72], which require scratchpad memory: CCB reserves 8N wordlines for N-bit operands, while CoMeFa uses 5N wordlines with the "One Operand Outside RAM (OOOR)" technique. PiCaSO needs only 4N wordlines, avoiding operand copying to the same bitline as CoMeFa. In the widest mode of a Virtex 36Kb BRAM, each PE in CCB and CoMeFa has 256 bits of register file storage, while PiCaSO has 1,024 bits. Figure 17 shows their memory utilization efficiency, which drops at higher precisions for CCB and CoMeFa. For 16-bit operands, CCB and CoMeFa have 50% and 68.8% efficiency, respectively, while PiCaSO achieves 93.8% efficiency. This demonstrates PiCaSO's superior memory efficiency.

8.3 GEMV Engine Implementation and Analysis

8.3.1 IMAGine GEMV Tile. Before implementing the GEMV tile, its components were individually analyzed to ensure they met design requirements. The GEMV tile includes a 12 × 2 PIM array and a two-stage pipeline in the fanout tree, optimized for Alveo U55 FPGA. Table 7 presents the utilization and performance of these components, along with their relative utilization in a GEMV tile. The controller and fanout network met timing constraints at up to 890 MHz. The PIM array, limited by BRAM's Fmax, achieved 737 MHz, matching the BRAM Fmax on Alveo U55. As shown in Table 7, the controller uses about 5% of the logic resources and no DSPs, while the PIM array consumes around 95%. Thus, the performance and scalability of the GEMV tile depend primarily on the PIM array, which is the desired outcome.

Table 8. Representatives of Virtex-7 and UltraScale+ [33]

Device	Tech	BRAM#	Ratio[a]	Max PE#[b]	ID
xcu55c-fsvh	US+	2,016	646	64K	U55
xc7vx330tffg	V7	750	272	24K	V7-a
xc7vx485tffg	V7	1,030	295	32K	V7-b
xc7v2000tfhg	V7	1,292	946	41K	V7-c
xc7vx1140tflg	V7	1,880	379	60K	V7-d
xcvu3p-ffvc	US+	720	547	23K	US-a
xcvu23p-vsva	US+	2,112	488	67K	US-b
xcvu19p-fsvb	US+	2,160	1,892	69K	US-c
xcvu29p-figd	US+	2,688	643	86K	US-d

[a]LUT-to-BRAM ratio.

[b]Number of PEs utilizing all BRAMs as PIMs.

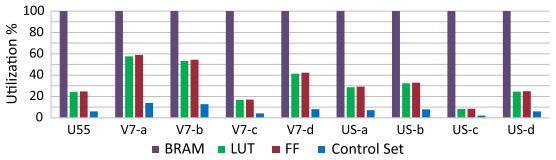

Fig. 18. Resource usage of the GEMV engine on representatives of Virtex-7 and UltraScale+ families utilizing 100% BRAMs as PIM overlays.

8.3.2 Scalability Study. To evaluate the scalability of the GEMV engine across different device families, we selected four representatives from AMD's Virtex-7 and UltraScale+ devices based on BRAM capacity and LUT-to-BRAM ratio. Table 8 lists these devices along with their BRAM capacity, LUT-to-BRAM ratio, and short IDs used in Figure 18. The target clock frequency was set to 100 MHz to avoid timing issues and only focus on logic utilization. Figure 18 presents a bar graph of post-implementation utilization for the GEMV engine, IMAGine, on these devices. IMAGine achieved 100% BRAM utilization as PIM overlays, providing 64K PEs in U55, with only 25% logic and 6% control set utilization. It scaled up to full BRAM capacity across all selected Virtex-7 and UltraScale+ devices.

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.

8.3.3 System-Level Timing Optimizations. Implementing the GEMV engine with all BRAMs as PIMs at the BRAM Fmax of 737 MHz on Alveo U55 required several iterations of device-specific optimization. Initially, we used a 4×4 PIM array without a fanout tree between the controller and the array. After running the default Vivado flow and making a few adjustments, the setup slack was -0.52 ns, with critical paths in the tile controller at pipeline stage A shown in Figure 9(a). To address this, we enabled pipeline stage A and proceeded with the second iteration. This iteration resulted in a setup slack of -0.38 ns, with timing issues caused by high fanout and long routes of control signals between the controller and PIM array. To resolve this, we synthesized a fanout tree with 2 levels and a fanout of 4 for the next iteration.

In the third iteration, the design achieved a setup slack of -0.27 ns. Further analysis revealed that many critical paths crossed hardened blocks on the Alveo U55, such as PLLs, PCIe, and CMAC blocks. Figure 19(a) shows some of the critical nets highlighted in white crossing a CMAC block. To address this, we implemented floorplanning blocks (pblocks) shown in Figure 19(b) for each tile, to confine the logic and routing within dedicated regions avoiding hard blocks. This required a 12×2 PIM array configuration for GEMV tiles. Figure 19(c) shows the placement and routing result in the final iteration, where high fanout control signals no longer crossed the CMAC block. Only the logically essential nets for east-to-west accumulation (highlighted in yellow), cross the CMAC block requiring minimal routing resources.

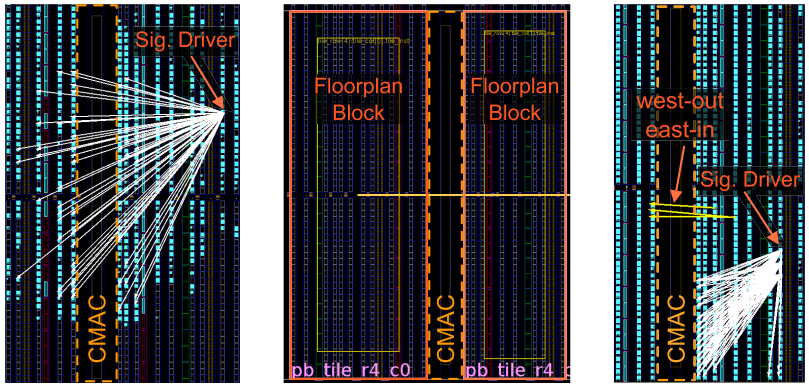

Fig. 19. Avoiding unnecessary hard-block (CMAC) crossing (a) placement and net connections before floorplanning, (b) floorplanning to localize logic and routing, and (c) placement and net connections in the final implementation.

The final design met the timing requirement at a 737 MHz clock, utilizing 100% available BRAMs as PIM blocks. Surprisingly, this clock rate exceeds that of custom GEMM accelerator ASICs like TPU v1-v2 (700 MHz) and Alibaba Hanguang 800 (700 MHz). Both Alveo U55 and TPU v2 are manufactured at 16 nm and Hanguang 800 at 12 nm technology nodes. So, this clock improvement is not due to a technology node difference. IMAGine on Alveo U55 features 64K PEs, the same as TPU v1 and four times that of TPU v2 (16K). However, it can only deliver up to 0.33 TOPS, significantly lower than TPU v1 (92 TOPS) and v2 (46 TOPS) due to its bit-serial architecture.

8.4 GEMV Engine Utilization and Performance Comparison

Table 9 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 [30, 31]. Though RIMA is specialized for accelerating RNNs, a major part of the system implements GEMV operation using **Dot Product Engines (M-DPEs)** [24]. The RIMA numbers are taken from Table 2 of [24] for comparison, which was evaluated on a Stratix 10 GX2800 FPGA with a BRAM Fmax of 1 GHz [73]. Its fastest reported configuration (RIMA-Fast) runs at 455 MHz, which is 2.2× slower than the BRAM Fmax. The largest reported configuration (RIMA-Large) utilizes 93% of BRAMs and runs at 278 MHz, 4× 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 [29]. Though CoMeFa-based designs run slightly faster than the CCB-GEMV engine, they are still roughly 3× 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.

SPAR-2 [26] 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 outperformed all existing designs. This is an important proof of concept design that dispels earlier beliefs that overlays cannot achieve BRAM Fmax clock frequencies at the system level [30, 31]. 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.

Table 9. Utilization and Frequency of PIM-Based GEMV/GEMM Engines

	LUT	FF	DSP	BRAM	f_{Sys}[a]	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[b]	10.1%	7.2%	0.0%	100.0%	737	100.0%

[a]System frequency in MHz.
[b]IMAGine with custom-BRAM implementation of PiCaSO.

As shown in Table 9, RIMA and CCB/CoMeFa-based GEMV engines exhaust either logic resources or DSPs, even with customized BRAM tiles. In contrast, IMAGine achieves a faster clock and better scalability, using no DSPs and only one-third of the device's logic resources. Like SPAR-2, IMAGine's bit-serial PEs eliminate the need for DSPs. With a custom-BRAM implementation (PiCaSO-CB), IMAGine would consume about 10% of device resources, making it fully scalable and suitable for resource-limited FPGAs.

8.5 GEMV Execution Latency

Figure 20 plots the GEMV latency for PIM designs, with matrix dimensions (square matrices) on the x-axis and latency in the log scale on the y-axis. Figure 20(b) shows execution times, calculated by multiplying latency in cycles by the corresponding clock periods of Table 9 system frequencies. We adopted the approach in [30] to model the block-level cycle latencies of CCB, CoMeFa, BRAMAC, and SPAR-2 using their analytical models. The global reduction tree of RIMA was modeled as an adder tree, perfectly pipelined with the last iteration of in-block accumulation. For SPAR-2, only the binary-add latency is shown, as the linear-add version is too slow to compare with the other systems.

As observed in Figure 20(a), BRAMAC has the shortest cycle latency, due to its hybrid bit-serial and bit-parallel MAC2 algorithm. MAC latency in BRAMAC increases linearly with operand bit-width, while it grows quadratically in bit-serial architectures like CCB, CoMeFa, SPAR-2, and PiCaSO. However, BRAMAC only supports 2, 4, and 8-bit precisions. BRAMAC [30] 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, due to their efficient reduction algorithm. IMAGine's cycle latency is shorter than SPAR-2 but longer than CCB/CoMeFa-based implementations. However, IMAGine operates at a clock speed at least 2× faster than the others, giving it the best overall execution time. Although CCB/CoMeFa GEMV engines use fewer cycles, IMAGine has significantly better performance due to its faster clock.

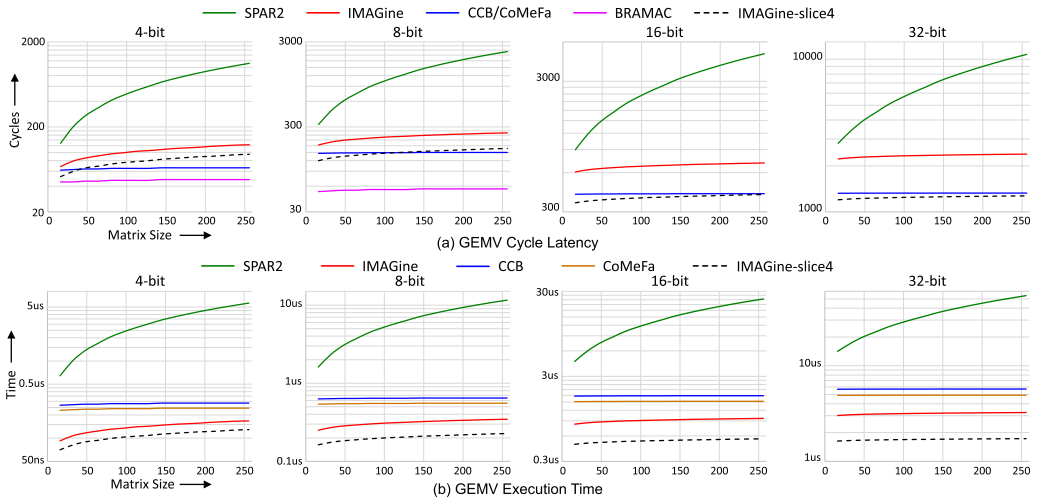

Fig. 20. Cycle latency and execution time of GEMV operation on different PIM array-based FPGA accelerators.

Table 10. Latency Breakdown of Multicycle Instructions of VV-Engine

Instr.	Cycles	Breakdown
vv_add	4	RF > RF > ADD > OREG
vv_sub	4	RF > RF > SUB > OREG
vv_mult	5	RF > RF > MULT > MULT > OREG
vv_activ	3	RF > RF > OREG (lookup)
vv_mov	3	RF > RF > OREG/SREG/ACT

Table 11. Utilization of 12 × 1 VTile Components on Alveo U55 at 737 MHz

	Ctrl	Tile Rel.	Fanout	Tile Rel.	VBlk-Arr	Tile Rel.	Tile
LUT	27	1.7%	0	0.0%	1,580	98.3%	1,607
FF	116	8.5%	247	18.1%	1,004	73.4%	1,367
DSP	0	0.0%	0	0.0%	12	100.0%	12
BRAM	0	0.0%	0	0.0%	12.0	100.0%	12

8.6 VTile

Before integrating the VV-Engine with the GEMV Engine to build DA-VinCi, a single VTile was studied to ensure it met design requirements. A 12 × 1 tile dimension was selected for implementation on the Alveo U55, matching the height of the GEMV tiles to simplify their connections. The VTile includes two pipeline stages in the fanout tree between the controller and VBlock array, matching the GEMV tile's instruction propagation latency, thus simplifying synchronization between the VV-Engine and GEMV Engine.

Table 10 shows the cycle count for multicycle instructions of the VV-Engine, with the Breakdown column indicating the execution sequence through Figure 12(a) and 13 blocks. Reading from the BRAM (RF) always takes 2 cycles due to its OREG s. The multiplier requires 2 cycles for fixed-point multiplication. The output of ALU is always stored in the OREG. Data movement between special registers—OREG, SREG, ACT—takes 1 cycle (not shown in the table) while transferring data from the register file to any special-purpose register takes 3 cycles.

Table 11 provides a detailed breakdown of the resource utilization of a 12 × 1 VTile on the Alveo U55 at 737 MHz, the BRAM Fmax. Comparing these utilization figures with the utilization of GEMV tile in Table 7 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 compared to the GEMV tile controller due to a simpler design without multicycle FSMs. Moreover, the VTile's fanout tree has lower flip-flop utilization due to fewer control signals. 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 and activation lookup-table storage, resulting

Table 12. Utilization of the Front-End Interface on Alveo U55

	Util.	Util.%	Available
LUT	50	0.004%	1,303,680
FF	32	0.001%	2,607,360
DSP	0	0.000%	9,024
BRAM	1	0.050%	2,016

Table 13. Utilization Breakdown of the Largest DA-VinCi Implementation (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	463,286	95.9%	19,703	4.1%	482,989	37.0%
FF	608,812	97.4%	16,202	2.6%	625,014	24.0%
DSP	0	0.0%	144	100.0%	144	1.6%
BRAM	1,871	92.8%	144	7.1%	2,016	100.0%

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.

8.7 DA-VinCi System-Level Analysis

8.7.1 Front-End Interface. Table 12 shows the front-end interface utilization on the Alveo U55 at BRAM Fmax. The "Available" column lists the total resources, while the "Util.%" column shows the device-level utilization. As discussed in Section 7.1, the simple design of the interface leads to minimal LUT and flip-flop usage, with 0 DSP utilization. FIFO-in and FIFO-out, used as the interface to the CPU, are implemented with 2 RAMB18, consuming one RAMB36 tile. Thus, the front-end interface module meets timing requirements and serves as a lightweight abstraction layer, ensuring portability and binary-code compatibility across future DA-VinCi implementations at minimal cost.

8.7.2 Compute Engines. Table 13 provides a detailed utilization breakdown of DA-VinCi and its compute engines. By utilizing 100% of the available BRAM, DA-VinCi offers 60K bit-serial PEs capable of performing parallel 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 13 with the IMAGine row in Table 9 reveals that system-level utilizations are almost identical, with the primary difference being the 1.6% DSP utilization to support the 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.

9 Application Latency Comparison

To evaluate DA-VinCi's performance on real-world deep-learning applications, it was compared with three classes of accelerators: the SPAR-2 benchmark overlay, PIM-based deep-learning FPGA accelerators, and custom FPGA accelerators. The deep-learning application benchmarks are selected from the published literature on these accelerators, with the comparison focusing on the inference latency.

9.1 Comparison with the Benchmark Overlay

Table 14 compares DA-VinCi with the benchmark overlay, SPAR-2, using the application benchmarks published in [27]. The LSTM-1 model, used for speech recognition on the TIMIT dataset [74], has an input size of 61, three hidden layers of size 250, and a fully-connected output layer of size 39. The LSTM-2 model, used for character recognition on the Shakespeare dataset [75], features an input size of 64, two hidden layers of size 128, and a fully-connected output layer of size 64. The MLP-1 model, designed for handwritten digit recognition on the MNIST dataset [76], includes 784

Table 14. Application Latency Comparison of DA-VinCi with the Benchmark Overlay SPAR-2 [27]

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	11,400	1	FxP32	133,890	56,207	313	0	200	US+
DA-VinCi	56.63	201.3	FxP32	521,991	675,180	2,250	250	737	US+
LSTM-2 (64, 128, 128, 64) on CharRec dataset									
SPAR-2	257.1	1	FxP16	133,890	56,207	313	0	200	US+
DA-VinCi	18.72	13.7	FxP16	141,095	178,127	640	128	737	US+
MLP-1 (784, 100, 100, 10) on MNIST dataset									
SPAR-2	300	1	FxP32	133,890	56,207	313	0	200	US+
DA-VinCi	22.02	13.6	FxP32	98,919	124,198	450	100	737	US+
GRU-1 (39, 256, 200, 10) on DeepSpeech dataset									
SPAR-2	2,100	1	FxP16	133,890	56,207	313	0	200	US+
DA-VinCi	17.59	119.4	FxP16	534,827	692,309	2,304	256	737	US+

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

inputs, two fully-connected layers of size 100, and an output layer of size 10. Lastly, the GRU-1 model, used for speech recognition on the DeepSpeech dataset [77], consists of an input size of 39, two hidden layers of sizes 256 and 200, and a fully-connected output layer of size 10.

As observed from Table 14, 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 is attributed to faster clock speed, more parallel MACs, and efficient reduction network design, at the cost of higher utilization of device resources. On the LSTM-2 benchmark, DA-VinCi achieved a 13.7× speed-up, reducing latency from 257.1 µs to 18.72 µs, 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 mainly 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. 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.

9.2 Comparison with Custom-BRAM PIM Accelerators

Table 15 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 [78] suite, as used in the original publications on RIMA [24] and CoMeFa [29]. The GEMM-1 kernel (m = 1,536, k = 512, n = 32) involves multiplying a weight matrix of size 1,536 × 512 with an input matrix of size 512 × 32. 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 = 1,024) benchmark includes a GRU layer with 1,024 hidden units.

As observed from Table 15, 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 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.

Table 15. Application Latency Comparison of DA-VinCi with Custom-BRAM PIM Accelerators

Design	Latency[a]	Speedup	Preci.[b]	Logic[d]	FF	BRAM	DSP	Freq[c]	FPGA
GEMM-1 (m = 1,536, k = 512, n = 32) from Deepbench									
CoMeFa [29]	388.99[e]	1	INT8	86,604	346,413	2,239	1,317	267	Arria 10
DA-VinCi	268.68	1.4	INT8	534,827	692,309	2,304	256	737	US+
LSTM-3 (h = 256) from DeepBench									
RIMA [24]	60	1	INT8	559,872	2,239,488	6,447	2,880	455	Stratix 10
DA-VinCi	2.52	23.8	INT8	534,827	692,309	2,304	256	737	US+
LSTM-4 (h = 512) from DeepBench									
RIMA [24]	20	1	INT8	690,509	2,762,036	8,088	2,880	417	Stratix 10
DA-VinCi	9.27	2.2	INT8	534,827	692,309	2,304	256	737	US+
GRU-2 (h = 1,024) from DeepBench									
RIMA [24]	2,630	1	INT8	653,184	2,612,736	7,619	2,880	417	Stratix 10
DA-VinCi	30.16	87.2	INT8	534,827	692,309	2,304	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 [29].

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.

9.3 Comparison with Custom FPGA Accelerators

Outperforming custom accelerators with a general overlay architecture is challenging. However, since DA-VinCi achieved BRAM Fmax as the clock speed and scaled to 100% of available BRAM, we expected competitive performance compared to custom FPGA accelerators. Table 16 compares DA-VinCi's latency and utilization with state-of-the-art custom deep-learning accelerators implemented on FPGAs. The LSTM-5 model was used to predict the real-time response of **High-Rate (HRate)** dynamic systems on the DROPBEAR dataset in [59]. It consists of an input size of 16 and three hidden layers of size 15. LSTM-2 is the same model included in Table 14, which was used for character recognition on the Shakespeare dataset using a Streaming accelerator (Stream) in [61]. The MLP-2 model was used by CM in [58] 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. GRU-1 is the same model included in Table 14. It was used for speech recognition on the DeepSpeech dataset using the DRNN accelerator with an optimal delta threshold of 0x80 in [56]. Lastly, the LSTM-6 model is from the DeepBench suite, with an LSTM layer of 1,024 hidden units. This was used by the custom accelerator Spartus [57] for speech recognition on the TIMIT dataset.

As observed from Table 16, 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 model. While bit-serial computing is slower compared to bit-parallel, DA-VinCi compensates for this with tens of thousands of bit-serial PEs running in parallel. However, because the model is

Table 16. Application Latency Comparison of DA-VinCi with Custom FPGA 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 [59]	4.72	1	FxP16	25,346	31,136	16	224	375	US+	HLS
DA-VinCi	11.8	0.4	FxP16	5,397	6,541	30	15	737	US+	Overlay
HRate [59]	2.49	1	FxP16	65,184	130,368	41	1,174	256	US+	HDL
DA-VinCi	11.8	0.2	FxP16	5,397	6,541	30	15	737	US+	Overlay
LSTM-2 (64, 128, 128, 64) on CharRec dataset										
Stream [61]	66	1	FxP16	95,263	118,261	259	1,095	420	US+	Overlay
DA-VinCi	18.72	3.5	FxP16	141,095	178,127	640	128	737	US+	Overlay
MLP-2 (784, 64, 64, 10) on MNIST dataset										
CM [58]	180	1	FxP8	18,218	11,670	222	6	100	V7	HLS
DA-VinCi	12.2	14.8	FxP8	39,524	47,700	192	64	540	V7	Overlay
GRU-1 (39, 256, 200, 10) on DeepSpeech dataset										
DRNN [56]	1,000[e]	1	FxP16	261,357	119,260	768	457.5	125	Z7	HDL
DA-VinCi	17.59	56.9	FxP16	534,827	692,309	2,304	256	737	US+	Overlay
LSTM-6 (h = 1,024) from DeepBench										
Spartus [57]	1	1	INT8	136,481	108,186	250	520	200	Z7	HDL
DA-VinCi	36.0	0.03	INT8	534,827	692,309	2,304	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 0×80 [56].

very small in these benchmarks, DA-VinCi loses its parallelism advantage and is hindered by its bit-serial architecture.

DA-VinCi achieved a 3.5× latency speed-up on the LSTM-2 benchmark compared to the Streaming accelerator [61], 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 CM [58] 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 56.9× speed-up compared to Delta-RNN [56] 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 is important to note that Delta-RNN is optimized for exploiting temporal dependencies in RNN inputs and activations, and the latency reported in Table 16 reflects its maximum speed with a delta-threshold of 0×80 [56]. 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 CBTD pruning [57]. 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 µs latency, whereas DA-VinCi required 36 µs, making DA-VinCi 36 times slower on this application 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.

10 Conclusion

PIM architectures have become popular frameworks replacing classic von Neumann architectures within domain-specific machine learning accelerators. In this article, we introduced DA-VinCi, a deep-learning accelerator overlay that leverages a PIM computing architecture to achieve high performance and scalability. Designed to run at the maximum frequency of BRAM (Fmax), DA-VinCi scales compute density linearly with the available BRAM resources on an FPGA exposing the total internal BRAM bandwidth of the device. The architecture is specifically optimized for low-latency inference in MLP, RNN, LSTM, and GRU networks, building on the highly customized PIM architecture, PiCaSO. Key features include a bit-serial binary hopping reduction network that reduces accumulation latency in GEMV operations. A scalable GEMV engine, IMAGine, was developed to ensure scalability across diverse FPGA families without compromising system clock speed below BRAM Fmax. Additionally, DA-VinCi incorporates a custom VV-Engine to accelerate point-wise operations and activation functions on post-GEMV resultant vectors. A lightweight front-end interface with an abstract instruction set was developed to ensure extensibility and application portability across future improvements and implementations of DA-VinCi.

Our analysis demonstrated significant improvements in utilization, performance, and reduction latency in the PIM block compared to a benchmark PIM overlay. The comparison of the PIM block with custom BRAM implementations highlighted that the PiCaSO architecture offers attractive design tradeoffs. Scalability studies showed that DA-VinCi scales linearly with increasing BRAM density, achieving impressive results on the Alveo U55 with 60K PEs, where it outperformed TPU v1-v2 and Alibaba Hanguang 800 in clock speed. Furthermore, comparative studies across multiple deep-learning applications revealed that DA-VinCi delivers up to 201× improvement over a baseline PIM overlay accelerator, 87× over existing PIM-based FPGA accelerators, and 57× over custom deep-learning accelerators on FPGAs. These results suggest that utilizing BRAMs on FPGAs as PIM blocks could significantly close the performance gap between FPGAs and ASICs in deep-learning applications, paving the way for more efficient and scalable deep-learning solutions.

References

[1] Young-Kyu Choi, Kisun You, Jungwook Choi, and Wonyong Sung. 2010 A real-time FPGA-based 20 000-word speech recognizer with optimized DRAM access. *IEEE Transactions on Circuits and Systems I: Regular Papers* 57, 8 (2010), 2119–2131.

[2] Dimitrios Danopoulos, Christoforos Kachris, and Dimitrios Soudris. 2018. Acceleration of image classification with caffe framework using FPGA. In *Proceedings of the 2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 1–4.

[3] Hongxiang Fan, Shuanglong Liu, Martin Ferianc, Ho-Cheung Ng, Zhiqiang Que, Shen Liu, Xinyu Niu, and Wayne Luk. 2018. A real-time object detection accelerator with compressed SSDLite on FPGA. In *Proceedings of the 2018 International Conference on Field-Programmable Technology (FPT)*, 14–21.

[4] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. 2017. ESE: Efficient speech recognition engine with sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 75–84.

[5] Cong Hao, Atif Sarwari, Zhijie Jin, Husam Abu-Haimed, Daryl Sew, Yuhong Li, Xinheng Liu, Bryan Wu, Dongdong Fu, Junli Gu, and Deming Chen. 2019. A hybrid GPU + FPGA system design for autonomous driving cars. In *Proceedings of the 2019 IEEE International Workshop on Signal Processing Systems (SiPS)*, 121–126.

[6] João Pedro Klock, Jhonatan Corrêa, Miguel Bessa, Janier Arias-Garcia, Felipe Barboza, and Carmo Meinertz. 2021. A new automated energy meter fraud detection system based on artificial intelligence. In *Proceedings of the 2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC)*, 1–8.

[7] Akira Kojima and Yohei Nose. 2018. Development of an autonomous driving robot car using FPGA. In *Proceedings of the 2018 International Conference on Field-Programmable Technology (FPT)*, 411–414.

[8] Minjae Lee, Kyuyeon Hwang, Jinhwan Park, Sungwook Choi, Sungho Shin, and Wonyong Sung. 2016. FPGA-based low-power speech recognition with recurrent neural networks. In *Proceedings of the 2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, 230–235.

[9] Peng Lv, Wei Liu, and Jinghui Li. 2020. A FPGA-based accelerator implementaion for YOLOv2 object detection using Winograd algorithm. In *Proceedings of the 2020 5th International Conference on Mechanical, Control and Computer Engineering (ICMCCE)*, 1894–1898.

[10] Duy Thanh Nguyen, Tuan Nghia Nguyen, Hyun Kim, and Hyuk-Jae Lee. 2019. A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 8 (2019), 1861–1873.

[11] Serkan Sağlam, Fatih Tat, and Salih Bayar. 2019. FPGA implementation of CNN algorithm for detecting malaria diseased blood cells. In *Proceedings of the 2019 International Symposium on Advanced Electrical and Communication Technologies (ISAECT)*, 1–5.

[12] D. Selvathi and R. Deiva Nayagam. 2016. FPGA implementation of on-chip ANN for breast cancer diagnosis. *Intelligent Decision Technologies* 10, 4 (2016), 341–352.

[13] Dong-Fong Syu, Su-Wei Syu, Shanq-Jang Ruan, Yu-Chang Huang, and Chuan-Kai Yang. 2015. FPGA implementation of automatic speech recognition system in a car environment. In *Proceedings of the 2015 IEEE 4th Global Conference on Consumer Electronics (GCCE)*, 485–486.

[14] Jin Wang and Shenshen Gu. 2021. FPGA implementation of object detection accelerator based on Vitis-AI. In *Proceedings of the 2021 11th International Conference on Information Science and Technology (ICIST)*, 571–577.

[15] Hao Xiao, Kaikai Zhao, and Guangzhu Liu. 2021. Efficient hardware accelerator for compressed sparse deep neural network. *IEICE Transactions on Information and Systems* E104.D, 5 (2021), 772–775.

[16] Ahmed J. Abd El-Maksoud, Mohamed Ebbed, Ahmed H. Khalil, and Hassan Mostafa. 2021. Power efficient design of high-performance convolutional neural networks hardware accelerator on FPGA: A case study with GoogLeNet. *IEEE Access* 9 (2021), 151897–151911.

[17] Huaixiang Hu, Jiatong Li, Chunchun Wu, Xueyang Li, and Yuping Chen. 2022. Design and implementation of intelligent speech recognition system based on FPGA. *Journal of Physics: Conference Series* 2171, 1 (January 2022), 012010.

[18] Safa Bouguezzi, Hana Ben Fredj, Tarek Belabed, Carlos Valderrama, Hassene Faiedh, and Chokri Souani. 2021. An efficient FPGA-based convolutional neural network for classification: Ad-MobileNet. *Electronics* 10, 18 (2021), 2272.

[19] Siyu Xiong, Guoqing Wu, Xitian Fan, Xuan Feng, Zhongcheng Huang, Wei Cao, Xuegong Zhou, Shijin Ding, Jinhua Yu, Lingli Wang, et al. 2021. MRI-based brain tumor segmentation using FPGA-accelerated neural network. *BMC Bioinformatics* 22, 1 (2021), 421.

[20] Ji-Hoon Kim, Juhyoung Lee, Jinsu Lee, Hoi-Jun Yoo, and Joo-Young Kim. June. 2020. Z-PIM: An energy-efficient sparsity aware processing-in-memory architecture with fully-variable weight precision. In *Proceedings of the 2020 IEEE Symposium on VLSI Circuits*. IEEE, 1–2.

[21] Bo Zhang, Shihui Yin, Minkyu Kim, Jyotishman Saikia, Soonwan Kwon, Sungmeen Myung, Hyunsoo Kim, Sang Joon Kim, Jae-Sun Seo, and Mingoo Seok. 2023. PIMCA: A programmable in-memory computing accelerator for energy-efficient DNN inference. *IEEE Journal of Solid-State Circuits* 58, 5 (2023), 1436–1449.

[22] Chia-Fu Lee, Cheng-Han Lu, Cheng-En Lee, Haruki Mori, Hidehiro Fujiwara, Yi-Chun Shih, Tan-Li Chou, Yu-Der Chih, and Tsung-Yung Jonathan Chang. 2022. A 12nm 121-TOPS/W 41.6-TOPS/mm2 all digital full precision SRAM-based compute-in-memory with configurable bit-width for AI edge applications. In *Proceedings of the 2022 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. IEEE, 24–25.

[23] Yongkee Kwon, Guhyun Kim, Nahsung Kim, Woojae Shin, Jongsoon Won, Hyunha Joo, Haerang Choi, Byeongju An, Gyeongcheol Shin, Dayeon Yun, et al. 2023. Memory-centric computing with SK Hynix's domain-specific memory. In *Proceedings of the 2023 IEEE Hot Chips 35 Symposium (HCS)*, 1–26.

[24] Xiaowei Wang, Vidushi Goyal, Jiecao Yu, Valeria Bertacco, Andrew Boutros, Eriko Nurvitadhi, Charles Augustine, Ravi R. Iyer, and Reetuparna Das. 2021. Compute-capable block RAMs for efficient deep learning acceleration on FPGAs. In *Proceedings of the 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 88–96.

[25] Suhail Basalama, Atiyehsadat Panahi, Ange-Thierry Ishimwe, and David Andrews. 2020. SPAR-2: A SIMD processor array for machine learning in IoT devices. In *Proceedings of the 2020 3rd International Conference on Data Intelligence and Security (ICDIS)*. IEEE, 141–147. DOI: https://doi.org/10.1109/ICDIS50059.2020.00025

[26] Atiyehsadat Panahi, Suhail Balsalama, Ange-Thierry Ishimwe, Joel Mandebi Mbongue, and David Andrews. 2021. A customizable domain-specific memory-centric FPGA overlay for machine learning applications. In *Proceedings of the 2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 24–27.

[27] Atiyehsadat Panahi. 2022. *A Memory-Centric Customizable Domain-Specific FPGA Overlay for Accelerating Machine Learning Applications*. Ph.D. Dissertation, University of Arkansas.

[28] A. Arora, T. Anand, A. Borda, R. Sehgal, B. Hanindhito, J. Kulkarni, and L. K. John. 2022. CoMeFa: Compute-in-memory blocks for FPGAs. In *Proceedings of the 2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1–9.

[29] Aman Arora, Atharva Bhamburkar, Aatman Borda, Tanmay Anand, Rishabh Sehgal, Bagus Hanindhito, Pierre-Emmanuel Gaillardon, Jaydeep Kulkarni, and Lizy K. John. 2023. CoMeFa: Deploying compute-in-memory on FPGAs for deep learning acceleration. *ACM Transactions on Reconfigurable Technology and Systems* 16, 3 (2023), 1–34.

[30] Yuzong Chen and Mohamed S. Abdelfattah. 2023. BRAMAC: Compute-in-BRAM architectures for multiply-accumulate on FPGAs. In *Proceedings of the 2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 52–62.

[31] Yuzong Chen, Jordan Dotzel, and Mohamed S. Abdelfattah. 2023. M4BRAM: Mixed-precision matrix-matrix multiplication in FPGA block RAMs. arXiv:2311.02758. Retrieved from http://arxiv.org/abs/2311.02758

[32] M. D. Arafat Kabir, Joshua Hollis, Atiyehsadat Panahi, Jason Bakos, Miaoqing Huang, and David Andrews. 2023.Making BRAMs compute: Creating scalable computational memory fabric overlays. In *Proceedings of the 2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 224–224.

[33] M. D Arafat Kabir, Ehsan Kabir, Joshua Hollis, Eli Levy-Mackay, Atiyehsadat Panahi, Jason Bakos, Miaoqing Huang, and David Andrews. 2023. FPGA processor in memory architectures (PIMs): Overlay or overhaul? In *Proceedings of the 2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 109–115.

[34] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 1–12.

[35] M. D. Arafat Kabir, Tendayi Kamucheka, Nathaniel Fredricks, Joel Mandebi, Jason Bakos, Miaoqing Huang, and David Andrews. 2024. IMAGine: An In-Memory Accelerated GEMV Engine Overlay. Retrieved from https://github.com/Arafat-Kabir/IMAGine

[36] M. D. Arafat Kabir, Tendayi Kamucheka, Nathaniel Fredricks, Joel Mandebi, Jason Bakos, Miaoqing Huang, and David Andrews. 2024. IMAGine: An in-memory accelerated GEMV engine overlay. In *Proceedings of the 2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE Computer Society, 220–226. DOI: https://doi.org/10.1109/FPL64840.2024.00038

[37] M. D. Arafat Kabir, Tendayi Kamucheka, Nathaniel Fredricks, Joel Mandebi, Jason Bakos, Miaoqing Huang, and David Andrews. 2024. DA-VinCi: A Deeplearning Accelerator Overlay Using In-Memory Computing. Retrieved from https://github.com/Arafat-Kabir/DA-VinCi

[38] William H. Kautz. 1969. Cellular logic-in-memory arrays. *IEEE Transactions on Computers* C-18, 8 (1969), 719–727. DOI: https://doi.org/10.1109/T-C.1969.222754

[39] Harold S. Stone. 1970. A logic-in-memory computer. *IEEE Transactions on Computers* C-19, 1 (1970), 73–78. DOI: https://doi.org/10.1109/TC.1970.5008902

[40] Yu Huang, Long Zheng, Pengcheng Yao, Jieshan Zhao, Xiaofei Liao, Hai Jin, and Jingling Xue. 2020. A heterogeneous PIM hardware-software co-design for energy-efficient graph processing. In *Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 684–695. DOI: https://doi.org/10.1109/IPDPS47924.2020.00076

[41] Christopher Wolters, Xiaoxuan Yang, Ulf Schlichtmann, and Toyotaro Suzumura. 2024. Memory is all you need: An overview of compute-in-memory architectures for accelerating large language model inference. arXiv:2406.08413. Retrieved from https://arxiv.org/abs/2406.08413

[42] Jack D. Kendall and Suhas Kumar. 2020. The building blocks of a brain-inspired computer. *Applied Physics Reviews* 7, 1 (2020), 011305.

[43] Geraldo Francisco Oliveira, Juan Gómez-Luna, Saugata Ghose, Amirali Boroumand, and Onur. Mutlu. 2022. Accelerating neural network inference with processing-in-dram: From the edge to the cloud. *IEEE Micro* 42, 06 (2022), 25–38. DOI: https://doi.org/10.1109/MM.2022.3202350

[44] Maya B. Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in memory: The Terasys massively parallel PIM array. *Computer* 28, 4 (1995), 23–31. DOI: https://doi.org/10.1109/2.375174

[45] Duncan G. Elliott, W. Martin Snelgrove, and Michael Stumm. 1992. Computational RAM: A memory-SIMD hybrid and its application to DSP. In *Proceedings of the 1992 Proceedings of the IEEE Custom Integrated Circuits Conference*, 30.6.1–30.6.4. DOI: https://doi.org/10.1109/CICC.1992.591879

[46] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. A case for intelligent RAM. *IEEE Micro* 17, 2 (1997), 34–44. DOI: https://doi.org/10.1109/40.592312

[47] David Patterson, Krste Asanovic, Aaron Brown, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Christoforos Kozyrakis, David Martin, Stylianos Perissakis, et al. 1997. Intelligent RAM (IRAM): The industrial setting,

applications, and architectures. In *Proceedings of the International Conference on Computer Design VLSI in Computers and Processors*, 2–7. DOI: https://doi.org/10.1109/ICCD.1997.628842

[48] Christoforos E. Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanovic, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton et al. 1997. Scalable processors in the billion-transistor era: IRAM. *Computer* 30, 9 (1997), 75–78. DOI: https://doi.org/10.1109/2.612252

[49] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system. *IEEE Access* 10 (2022), 52565–52608. DOI: https://doi.org/10.1109/ACCESS.2022.3174101

[50] Fabrice Devaux. 2019. The true processing in memory accelerator. In *Proceedings of the 2019 IEEE Hot Chips 31 Symposium (HCS)*, 1–24. DOI: https://doi.org/10.1109/HOTCHIPS.2019.8875680

[51] Jin Hyun Kim, Shin-Haeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, Bengseng Phuah, et al. 2021. Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for ML accelerators and beyond. In *Proceedings of the 2021 IEEE Hot Chips 33 Symposium (HCS)*, 1–26. DOI: https://doi.org/10.1109/HCS52781.2021.9567191

[52] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. 2018. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 383–396.

[53] Xiaowei Wang, Vidushi Goyal, Jiecao Yu, Valeria Bertacco, Andrew Boutros, Eriko Nurvitadhi, Charles Augustine, Ravi Iyer, and Reetuparna Das. 2021. Compute-capable block RAMs for efficient deep learning acceleration on FPGAs. In *Proceedings of the 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 88–96.

[54] Aman Arora, Tanmay Anand, Aatman Borda, Rishabh Sehgal, Bagus Hanindhito, Jaydeep Kulkarni, and Lizy K. John. 2022. CoMeFa: Compute-in-memory blocks for FPGAs. In *Proceedings of the 2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 1–9.

[55] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL™ deep learning accelerator on arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 55–64.

[56] Chang Gao, Daniel Neil, Enea Ceolini, Shih-Chii Liu, and Tobi Delbruck. 2018. DeltaRNN: A power-efficient recurrent neural network accelerator. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 21–30.

[57] Chang Gao, Tobi Delbruck, and Shih-Chii Liu. 2024. Spartus: A 9.4 TOp/s FPGA-based LSTM accelerator exploiting spatio-temporal sparsity. *IEEE Transactions on Neural Networks and Learning Systems* 35, 1 (2024), 1098–1112.

[58] Ehsan Kabir, Arpan Poudel, Zeyad Aklah, Miaoqing Huang, and David Andrews. 2022. A runtime programmable accelerator for convolutional and multilayer perceptron neural networks on FPGA. In *Proceedings of the International Symposium on Applied Reconfigurable Computing*. Springer, 32–46.

[59] Ehsan Kabir, Daniel Coble, Joud N. Satme, Austin R. J. Downey, Jason D. Bakos, David Andrews, and Miaoqing Huang. 2023. Accelerating LSTM-based high-rate dynamic system models. In *Proceedings of the 2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 327–332.

[60] Austin Downey, Jonathan Hong, Jacob Dodson, Michael Carroll, and James Scheppegrell. 2020. Millisecond model updating for structures experiencing unmodeled high-rate dynamic events. *Mechanical Systems and Signal Processing* 138 (2020), 106551.

[61] Lenos Ioannou and Suhaib A. Fahmy. 2023. Streaming overlay architecture for lightweight LSTM computation on FPGA SoCs. *ACM Transactions on Reconfigurable Technology and Systems* 16, 1 (2023), 1–26.

[62] M. D. Arafat Kabir, Ehsan Kabir, Joshua Hollis, Eli Levy-Mackay, Atiyehsadat Panahi, Jason Bakos, Miaoqing Huang, and David Andrews. 2023. PiCaSO: A Scalable and Fast PIM Overlay. Retrieved from https://github.com/Arafat-Kabir/PiCaSO

[63] X. Wang, V. Goyal, J. Yu, V. Bertacco, A. Boutros, E. Nurvitadhi, C. Augustine, R. Iyer, and R. Das. 2021. Compute-capable block RAMs for efficient deep learning acceleration on FPGAs. In *Proceedings of the 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 88–96.

[64] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization, 2004 (CGO '04)*. IEEE, 75–86. Retrieved from https://llvm.org/docs/index.html

[65] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14. Retrieved from https://mlir.llvm.org/

[66] M. D. Arafat Kabir. 2024. DA-VinCi IR3 Assember (DavinciAsm) Version 0.1. Retrieved from https://github.com/Arafat-Kabir/DA-VinCi/blob/master/work/scripts/davinci_assembler.py
[67] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (1986), 533–536.
[68] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. Retrieved from http://www.deeplearningbook.org
[69] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.
[70] Alex Graves, Navdeep Jaitly, and Abdel-Rahman Mohamed. 2013. Hybrid speech recognition with deep bidirectional LSTM. In *Proceedings of the 2013 IEEE Workshop on Automatic Speech Recognition and Understanding*. IEEE, 273–278.
[71] Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. 2017. FPGA-based accelerator for long short-term memory recurrent neural networks. In *Proceedings of the 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 629–634.
[72] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. 2018. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *Proceedings of the 2018 ACM/IEEE 45Th Annual International Symposium on Computer Architecture (ISCA)*, 383–396.
[73] Intel. 2022. Intel® Stratix® 10 Device Datasheet. Retrieved from https://www.intel.com/content/www/us/en/docs/programmable/683181/current/memory-block-specifications.html
[74] John W. Lyons. 1993. *DARPA TIMIT: Acoustic-Phonetic Continuous Speech Corpus*. National Institute of Standards and Technology.
[75] TensorFlow. 2017. Shakespear Dataset. Retrieved from https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt
[76] Yann LeCun, Corinna Cortes, Chris Burges. 2010. MNIST handwritten digit database. Retrieved from https://ieeexplore.ieee.org/document/6296535
[77] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. 2014. Deep speech: Scaling up end-to-end speech recognition. arXiv:1412.5567. Retrieved from https://arxiv.org/abs/1412.5567
[78] Baidu Research. 2016. Baidu DeepBench. Retrieved from https://svail.github.io/DeepBench/

Received 6 March 2025; revised 29 August 2025; accepted 22 September 2025