Hardware-efficient matrix multiplication core optimization for edge AI on FPGA

Phan Hong Minh[1*], Nguyen Tien Viet[2], Do Doanh Dien[1]

[1]Institute of Information Technology and Electronics, Academy of Military Science and Technology, 17 Hoang Sam, Nghia Do, Hanoi, Vietnam;
[2]Information Technology Office, Military Region 2, 3268 Hung Vuong, Van Phu, Phu Tho, Vietnam.
[*]Corresponding author: phanhongminh1979@gmail.com
Received 4 Aug. 2025; Revised 23 Sep. 2025; Accepted 10 Oct. 2025; Published 30 Oct. 2025.
DOI: https://doi.org/10.54939/1859-1043.j.mst.IITE.2025.123-130

ABSTRACT

This paper presents an optimization approach for matrix multiplication IP cores on FPGA by transforming convolution operations into matrix multiplications. The proposed method leverages parallel computation combined with simultaneous data loading within the same processing cycle, thereby reducing memory requirements and computational latency. Furthermore, casting the output data from 64-bit to 32-bit effectively shrinks the output buffer, resulting in significant hardware resource savings. Simulation results on ModelSim and Vivado–Vitis demonstrate that the design achieves higher computational efficiency and resource utilization compared to traditional implementations, while maintaining stable processing time. This work contributes to the design of CNN inference accelerators on FPGA for edge AI applications, where resource constraints and power consumption are critical factors..

Keywords: IP cores; Matrix multiplication; FPGA-CNN; MAC; Vivado-Vitis.

1. INTRODUCTION

Efficient implementation of convolutional neural networks (CNNs) on resource-constrained hardware platforms, particularly FPGAs, has been an active research area in recent years. Several notable works have explored different strategies to reduce computational complexity, optimize dataflow, and improve energy efficiency while maintaining model accuracy. Han et al. [2] introduced a weight pruning technique that reduces the number of parameters in neural networks without significantly affecting accuracy. This method enables efficient deployment of CNNs on hardware with limited memory and computation resources. Extending this idea, Wen et al. [3] proposed a structured sparsity learning approach that enforces group-level sparsity in weights, further reducing computation overhead and facilitating more hardware-friendly designs. Targeting FPGA implementation specifically, Gschwend [4] presented ZynqNet, a CNN accelerator optimized for embedded FPGA platforms, demonstrating that careful architectural design can achieve high performance under tight resource constraints. In parallel, several studies have investigated algorithmic optimization techniques. For instance, Li et al. [5] adopted the Winograd convolution algorithm to reduce the number of arithmetic operations in CNNs, thereby improving energy efficiency on FPGAs. Building upon this, Liu et al. [6] proposed WinoCNN, a systolic array architecture that integrates Winograd-based kernel sharing, achieving higher throughput and better resource utilization for CNN inference. Similarly, Zhang et al. [7] designed a unified input Winograd architecture that allows layer-to-layer data reuse, significantly reducing data movement and improving performance in FPGA-based accelerators. Recently, to efficiently exploit sparsity directly in hardware, Taka et al. [8] proposed systolic sparse tensor slices (SST) - FPGA-native building blocks that support multiple sparsity levels, enabling higher operating frequency and reduced area compared to purely dense matrix computation circuits.

While the above works emphasize weight pruning, sparsity, and Winograd-based optimization, our research focuses on optimizing dataflow and parallelism on FPGA, reducing memory footprint,

and providing an efficient CNN acceleration solution with lower hardware resource usage for edge AI applications

2. PROPOSED ENHANCEMENTS AND OPTIMIZATION

2.1. Matrix convolution on FPGA hardware

- *System architecture diagram:*

The proposed matrix multiplier is designed to compute the product of two 32×32 matrices with 16-bit fixed-point elements. The design is developed in VHDL and targets both FPGA prototyping and potential ASIC integration. The top-level architecture includes three main functional components:

+ Matrix Processing Core (IntMatMulCore): Performs the actual multiplication and accumulation of matrix elements.

+ Dual-Port RAM Blocks (dpram1024x16 and dpram1024x64): Used to store input matrices and intermediate/output results.

+ Finite State Machine (FSM) Controller: Manages the control flow, synchronization, and memory addressing.

The architecture is modular and scalable, allowing future adaptation to different matrix sizes or higher precision [9-11].

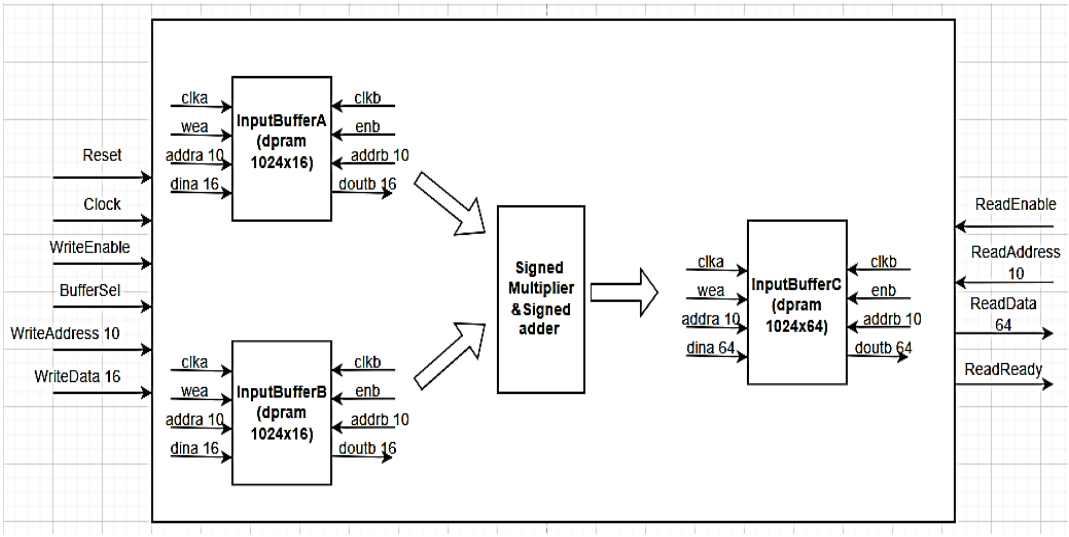

Figure 1. *Functional block diagram of matrix multiplier.*

- *Memory organization:*

InputBufferA and InputBufferB are both implemented using dual-port RAMs with the following specifications:

+ Address width: 10 bits (supporting 1024 elements).

+ Data width: 16-bit signed integers.

+ Access: Simultaneous read/write enabled via separate clka and clkb ports.

Each input matrix (A and B) contains 32×32 = 1024 elements, making 1024×16 RAM an optimal choice.

OutputBufferC is also a dual-port RAM used to store the result matrix:

+ Data width: 64 bits, sufficient to hold the sum of 32 multiplications without overflow.

+ Address width: 10 bits (supporting 1024 results).

This memory structure ensures efficient data flow and avoids resource bottlenecks during matrix computation.

- *Flowchart of the algorithm:*

Matrix A (32×32) and Matrix B (32×32) are first loaded into dual-port RAM blocks. The memory is organized such that each row of matrix A and each column of matrix B can be accessed in parallel or sequentially, depending on available hardware resources (figure 2a). Follow [11], The result matrix C is calculated (1) as:

$$C_{ij} = \sum_{k=0}^{31} A_{i,k}.B_{k,j} \quad (1)$$

2.2. Improved and optimized solutions

2.2.1. Buffer optimization for area efficiency

To improve area efficiency and reduce memory footprint, the input and output buffer sizes are optimized without compromising the correctness of the 32×32 matrix multiplication.

Input Buffers: From 16×1024 to 16×32.

Originally, both input matrices A and B were stored entirely in dual-port RAMs sized 1024×16 bits. However, since the matrix size is fixed at 32×32, storing all 1024 elements simultaneously is unnecessary if data is fed in a streaming or tiled fashion.

In the optimized design:

+ InputBufferA and InputBufferB are resized to 16×32, only holding a single row of A and a single column of B at a time.

+ The processing core accesses these rows/columns iteratively, reusing the buffer for each step of computation.

This significantly reduces BRAM utilization, especially important for FPGA designs.

Output Buffer: From 64×1024 to 32×1024.

For the result matrix C, each element accumulates the sum of 32 products. Instead of using a full 64-bit wide memory for all 1024 results, the output buffer is restructured as:

+ 32-bit partial sums per cycle, maintained across 32 cycles.

+ 32×1024 memory organization, allowing distributed accumulation over multiple rows/columns.

This change balances between storage width and memory depth, leading to a more efficient implementation.

Table 1. *Summary of buffer size changes.*

Component	Original size	Optimized size
InputBufferA	1024×16 bits	32x16 bits
InputBufferA	1024×16 bits	32x16 bits
OutputBufferC	1024×64 bits	1024x32 bits

By adopting this tiling and reuse strategy, the design reduces memory requirements and power consumption, enabling more compact and scalable implementations.

2.2.2. Output buffer width reduction via type casting (int64 to int32)

In the original design, the output buffer (OutputBufferC) was implemented with a 64-bit width to safely store the sum of 32 products of 16-bit integers (i.e., int16 × int16 × 32). While this ensures overflow protection, it doubles the I/O width compared to the 32-bit input buffers, leading to increased complexity in pin assignment, routing, and peripheral interfacing.

Optimized Approach: From int64 to int32.

To reduce I/O pins and simplify downstream data handling, the output data is cast from 64-bit to 32-bit signed integers (int32) before storage or transmission. This optimization is based on the observation that:

$$\max\left(\left|A_{i,k}.B_{k,j}\right|\right) = \left|(2^{15}-1)^2\right| = 32.767^2 \approx 2^{30}$$

and the total sum of 32 products:

$$C_{ij} = \sum_{k=0}^{31} A_{i,k}.B_{k,j} \leq 32.(2^{15}-1)^2 \approx 2^{30}.32 = 2^{35}$$

Assuming that the application domain (e.g., image processing, ML inference) tolerates 32-bit saturation or rounding, the output can be safely cast using the following logic:

$$C_{i,j}^{\text{int32}} \begin{cases} 2^{31}-1 \text{ if result } C_{ij} > 2^{31}-1 \\ -2^{31} \text{ if result } C_{ij} < -2^{31} \\ C_{ij} \text{ Otherwise} \end{cases} \quad (2)$$

This formula is equivalent to a clamping or saturation logic operation in hardware.

This optimization supports the use of a 32-bit output buffer and halves the number of necessary I/O pins.

Hardware Implications:

- Reduced I/O width: Fewer I/O pads for ASICs from 106 pads to 69 pads, and lower board complexity on FPGA.

- Simplified interfacing: Easier integration with 32-bit bus systems (AXI4-lite, TileLink, etc.).

- Lower power consumption: Due to reduced toggling and fewer active lines.

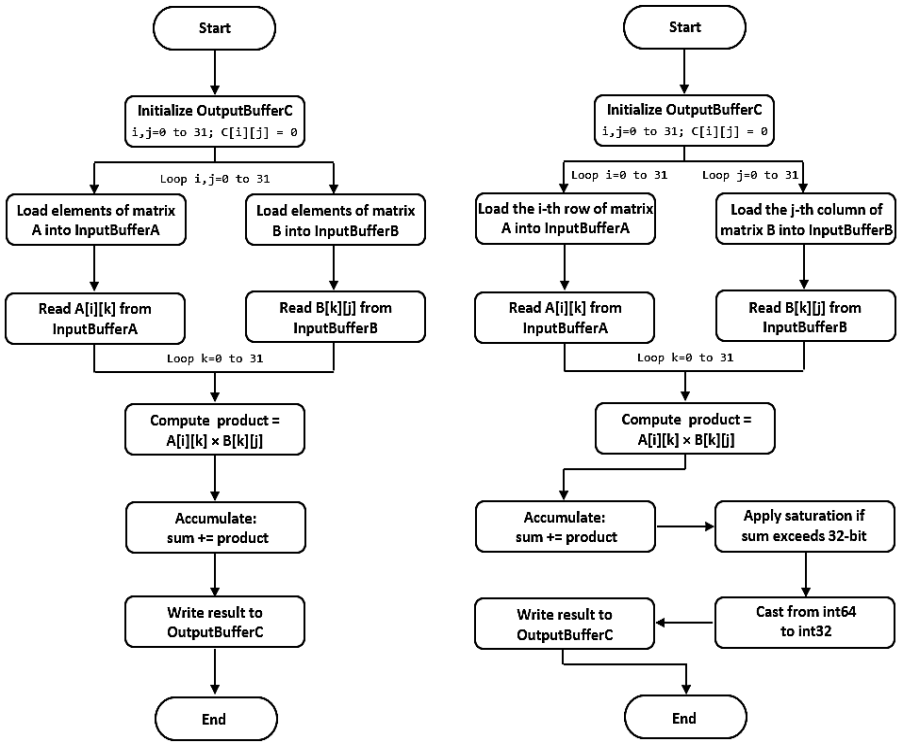

Figure 2. *Flowchart of the algorithm (a) and proposed with optimization (b).*

3. RESULTS AND DISCUSSION

3.1. Synthesizing RTL design

To evaluate the efficiency of the optimized matrix multiplication core, synthesis results before and after optimization were compared using Xilinx ISE14.7 and Vivado 2024. The analysis focuses on key metrics such as memory usage, logic resource consumption, and I/O requirements. The optimized design maintains the same total number of RAM blocks (3) as the original, but reorganizes their configurations more efficiently. Specifically, two 1024×16-bit RAMs and one 1024×64-bit RAM in the original design were replaced by one 1024×32-bit RAM and two 32×16-bit RAMs. This reorganization enables the use of a 32-bit output buffer and reduces the number of required I/O pins from 106 to 69, achieving a 35% reduction in external pin usage.

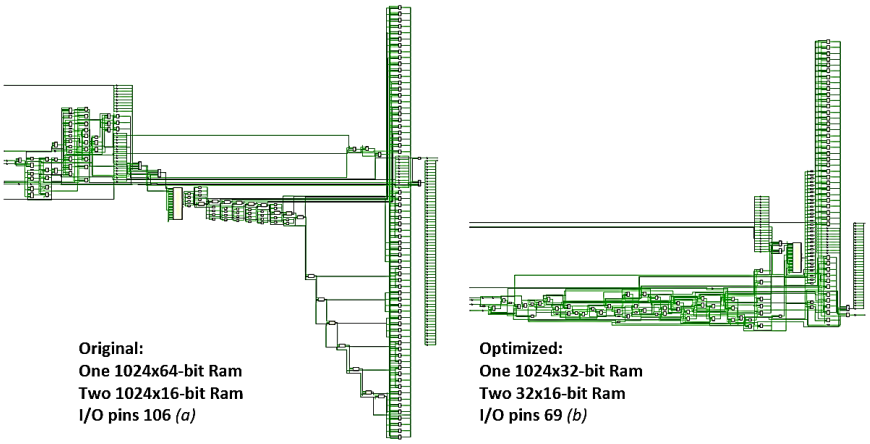

Figure 3. *Synthesizing hardware logic design (a) original and (b) optimized.*

Table 2. *Device utilization summary.*

No.	Component	Original	Optimized
1	1024x64-bit dual-port RAM	1	-
2	1024x16-bit dual-port RAM	2	-
3	1024x32-bit dual-port RAM	-	1
4	32x16-bit dual-port RAM	-	2
5	Slice LUTs	64	96
6	Slice Registers	84	76
7	Block Ram Tile	3	1
8	DSPs	1	1
13	LUT1	5	1
14	LUT2	42	7
15	LUT3	7	48
16	LUT4	4	7
17	LUT5	11	11
18	LUT6	6	9
19	RAMB18E1	2	-
20	RAM32M	-	6
21	RAMB36E1	2	1
22	FDRE (D flip-flop)	84	76
23	Top Cell	289	237
24	IOBuf - I/O Pins	106	69
25	Total power(mW)	35.52	18.457

The optimization process yielded a significantly more efficient FPGA design. While Slice LUTs saw an increase from 64 to 96 and LUT3s rose sharply from 7 to 48, indicating a potential redistribution of logic complexity, this was offset by a reduction in Slice Registers from 84 to 76 and substantial decreases in simpler logic gates, with LUT1s dropping from 5 to 1 and LUT2s from 42 to 7. Memory resource utilization improved markedly, as Block RAM Tiles were reduced from 3 to 1, RAMB18E1 instances were eliminated, and RAMB36E1 usage was halved, with the introduction of 6 RAM32M units. DSPs remained constant at 1. Overall, the Top Cell area decreased from 289 to 237, and critically, total power consumption was nearly halved from 35.52 mW to 18.457 mW, demonstrating a highly successful optimization in terms of both area and power efficiency.

3.2. Simulating RTL design

The testbench setup begins with clock generation, where the clock signal toggles from '0' to '1' every 5 ns, yielding a complete clock period of 10 ns. Matrix A is loaded from matrixA.txt and stored in the internal array matrixA, while matrix B is similarly read from matrixB.txt and stored in matrixB. For optimized access, matrix B is transposed into matrixB_T. Subsequently, each row of matrix A is sequentially written into RAM A with BufferSel = '1', and each row of the transposed matrix B is written into RAM B with BufferSel = '0', with a delay of one clock cycle between element writes.

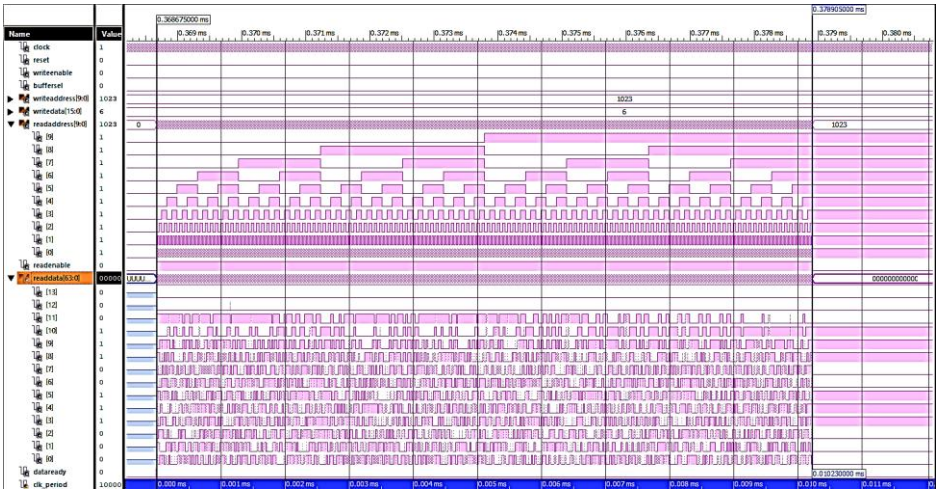

Figure 4. *Simulating RTL design original version.*

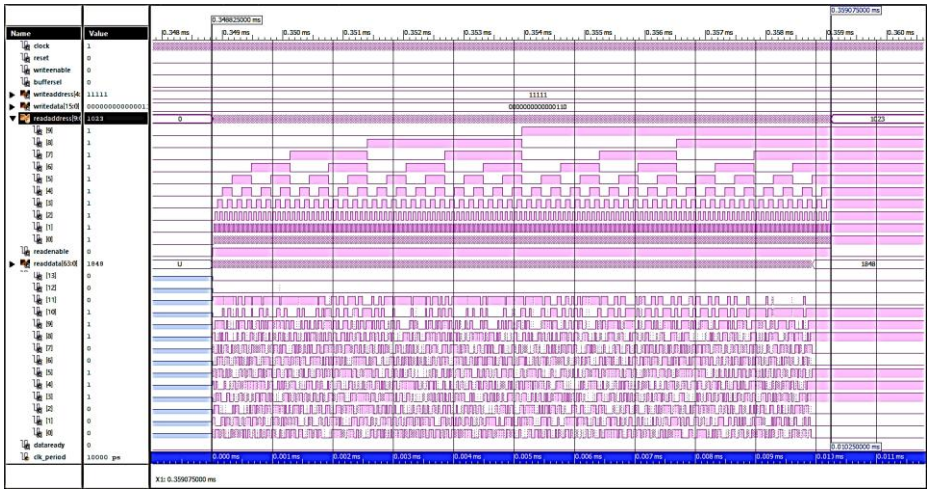

Figure 5. *Simulating the RTL design optimized version.*

Once the data loading is complete, the testbench waits for the DataReady signal to assert high (DataReady = '1') before proceeding to the result phase. Finally, all 32×32 elements of the result matrix C are read from the output interface using ReadAddress, and the complete matrix is written to matrixC.txt using the VHDL textio library.

RTL simulation with a clock period of t = 10 ns shows that the execution time is 0.01024 ms (figure 4). For the optimized version (figure 5), the execution time remains the same at 0.01024 ms. Thus, the optimization does not affect the overall execution time.

4. CONCLUSIONS

This work proposes an architectural optimization method for matrix convolution processing elements (PEs), based on structured matrix representation and buffer reorganization, aiming to efficiently implement convolutional neural networks (CNNs) on FPGA platforms. By reducing the number of RAM buffers and narrowing the output signal width from 64 bits to 32 bits, the design achieves significant reductions in memory usage and I/O pin count, thereby lowering hardware resource consumption and post-synthesis power usage.

However, the proposed approach introduces several limitations. The integration of finite state machines (FSMs) within each PE, along with a centralized FSM for dataflow control, increases system complexity in both design and verification phases.

Experimental results confirm that the proposed method significantly improves parallel processing performance and dataflow efficiency while reducing hardware resource usage and energy consumption. These results demonstrate the feasibility and effectiveness of the design for edge AI applications, which are constrained by strict limits on area, power, and I/O bandwidth. As a future direction, the design will be implemented on silicon using the FreePDK45nm process, targeting real-world deployment in intelligent edge devices.

REFERENCES

[1]. Nguyen, X.-Q. and Pham-Quoc, C., *"An FPGA-base Convolution IP Core for Deep Neural Networks Acceleration,"* Rev Journal on Electronics and Communications, Vol. 12, No. 1–2, pp. 1–6 (2022). DOI: 10.21553/rev-jec.286.

[2]. Han, S., Pool, J., Tran, J., and Dally, W. J., *"Learning Both Weights and Connections for Efficient Neural Networks,"* Neural Information Processing Systems (NeurIPS), Vol. 28 (2015).

[3]. Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H., *"Learning Structured Sparsity in Deep Neural Networks,"* Advances in Neural Information Processing Systems (NeurIPS) (2016).

[4]. Gschwend, D., *"ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network,"* arXiv Preprint, arXiv:2005.06892 (2020).

[5]. Li, Y., et al., *"Implementation of Energy-Efficient Fast Convolution Algorithm for Deep Convolutional Neural Networks Based on FPGA,"* Electronics Letters, Vol. 56, No. 5, pp. 234–236 (2020).

[6]. Liu, X et al., *"WinoCNN: Kernel Sharing Winograd Systolic Array for Efficient Convolutional Neural Network Acceleration on FPGAs,"* Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP) (2021).

[7]. Zhang, Y., et al., *"An Efficient Convolutional Neural Network Accelerator Design on FPGA Using the Layer-to-Layer Unified Input Winograd Architecture,"* Electronics, Vol. 14, No. 6, Article 1182 (2025). DOI: 10.3390/electronics14061182.

[8]. Taka, E., Huang, N.-C., Chang, C.-C., Wu, K.-C., Arora, A., and Marculescu, D., *"Systolic Sparse Tensor Slices: FPGA Building Blocks for Sparse and Dense AI Acceleration,"* arXiv Preprint, arXiv:2502.03763v1 [cs.AR] (2025).

[9]. https://www.fpga4student.com/2016/11/matrix-multiplier-core-design.html

[10]. https://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/f2020/bjd86_lgp36/bjd86_lgp36/index.html

[11]. https://www.mathworks.com/help/hdlverifier/xilinxfpgaboards/ug/large-matrix-multiplication-using-ethernet-aximaster.html

TÓM TẮT

Cải tiến và tối ưu hoá thuật toán của lõi IP nhân chập ma trận trong mạng nơ-ron trên FPGA

Bài báo trình bày một phương pháp tối ưu hoá lõi IP nhân ma trận trên FPGA thông qua việc biến đổi phép tích chập thành phép nhân ma trận. Giải pháp đề xuất khai thác cơ chế tính toán song song kết hợp nạp dữ liệu đồng thời trong cùng một chu kỳ xử lý, giúp giảm nhu cầu bộ nhớ và độ trễ tính toán. Ngoài ra, việc ép kiểu dữ liệu đầu ra từ 64-bit xuống 32-bit góp phần thu nhỏ bộ đệm kết quả, qua đó tiết kiệm đáng kể tài nguyên phần cứng. Kết quả mô phỏng trên ModelSim và Vivadol-Vitis cho thấy thiết kế đạt hiệu suất tính toán và hiệu quả tài nguyên vượt hơn so với các cách triển khai truyền thống, đồng thời vẫn đảm bảo thời gian tính toán ổn định. Công trình này hướng tới thiết kế chip tăng tốc suy luận CNN trên FPGA cho các ứng dụng AI biên, nơi hạn chế về tài nguyên và điện năng là các ràng buộc quan trọng.

Từ khoá: Lõi IP; Nhân ma trận; FPGA-CNN; Vivado-Vitis.