A Scalable External Memory Access and On-Chip Storage Architecture for Edge-AI Accelerators
– Multi-Path Rolling Data Refresh and Layer-Wise Bank Allocation –

Quan Cheng[1,2], Huizi Zhang[2], Qiufeng Li[2], Yuan Liang[2], Mingtao Zhang[1], Zhenzhe Chen[1], Ruilin Zhang[1], Jinjun Xiong[3], Mingqiang Huang[2], Longyang Lin[2], Masanori Hashimoto[1]*

[1]Department of Communications and Computer Engineering, Kyoto University, Kyoto, Japan
[2]School of Microelectronics, Southern University of Science and Technology, Shenzhen, China
[3]Department of Computer Science and Engineering, University at Buffalo, USA

***Abstract*—For resource-constrained AI accelerators applied in edge computing, achieving high power efficiency in neural network (NN) model computation is crucial. However, current designs often overlook the efficiency of off-chip/on-chip data interaction, leading to high latency, which in turn results in suboptimal power efficiency during computation. Additionally, inefficient memory bank allocation further exacerbates latency by causing underutilization of storage resources, thereby contributing to higher overall latency and energy consumption. To address these challenges, this paper proposes a scalable multi-path rolling data refresh and layer-wise bank allocation architecture. The rolling data refresh mechanism enables efficient data interaction between off-chip and on-chip storage, reducing latency and minimizing the area overhead of on-chip memories. The layer-wise bank allocation optimizes on-chip memory utilization according to specific application requirements, improving memory efficiency. A case study on a 28nm AI accelerator demonstrates a 30.6% reduction in area, achieves a power efficiency of 7.36–10.28 TOPS/W, and reduces external memory access by 2.63% to 37.24% on VGG16 and ViT-Small.**

***Index Terms*—AI accelerator, rolling data refresh, layer-wise bank allocation, power efficiency, edge computing.**

I. Introduction

In the era of edge computing, the need for energy-efficient AI accelerators has been more critical. Edge-AI applications, such as real-time video processing, autonomous navigation, and health monitoring, are often constrained by limited resources, including power, memory, and computational capabilities [1], [2]. To achieve high performance in these scenarios, accelerators must strike a delicate balance between computational efficiency and power consumption [3], [4]. However, current AI accelerators often fail to fully address the inherent challenges of data transfer and memory utilization, which significantly impact both latency and energy efficiency [5].

A key issue in many AI accelerator designs is the inefficient interaction between off-chip and on-chip memory [6]. As neural networks (NNs) become more complex, the volume of data exchange increases significantly, leading to substantial delays. The latency, directly impacts the overall throughput of the system, resulting in poor power efficiency. Furthermore, many existing designs overlook the need for efficient memory allocation, often resulting in underutilized or poorly allocated memory resources [7]–[9]. This inefficient use of memory further exacerbates latency and energy consumption, especially in resource-constrained systems where every cycle counts.

To address these challenges, this paper introduces a scalable architecture for the deployment of edge-AI accelerators. Highlights of this paper are:

- **Multi-Path Rolling Data Refresh**: The proposed refresh mechanism ensures that data transmission between off-chip and on-chip memory is seamless and streamlined. Using a circular data refresh approach, the system ensures a more consistent and efficient flow for data transmission, avoiding the additional overhead caused by read-write conflicts. Furthermore, this approach leads to a significant reduction (30.6% in a case study) in chip area overhead.
- **Layer-Wise Memory Bank Allocation**: The layer-wise memory bank allocation strategy further optimizes memory usage by dynamically assigning memory based on the specific needs of each NN layer. By adjusting memory resources in a layer-wise way, this targeted allocation reduces waste, enhances on-chip data reuse, and minimizes external memory access (up to 37.24% reduction in real NNs) while maintaining high throughput.

The following sections begin with a review of related AI accelerator designs, focusing on key challenges in data transfer and memory allocation (Section II). We then present our proposed architecture, featuring a rolling data refresh mechanism and dynamic bank allocation strategy (Section III). To evaluate our design, we deploy an edge-AI accelerator as a test platform (Section IV). Experimental results are then provided, demonstrating the performance of our approach and comparing it with state-of-the-art (SOTA) accelerators (Section V). Finally, we conclude the paper in Section VI.

II. Related Work

Edge-AI applications often face resource constraints, requiring careful balance between performance and power consumption. Many recent efforts have focused on improving memory efficiency, reducing latency, and optimizing power consumption through different memory access architectures and dynamic resource allocation strategies.

Symons et al. introduced a loop-order-based memory allocation (LOMA) method [8], which schedules DNN workloads on accelerators by leveraging the nested loop structure of DNN layers. LOMA improves memory utilization, supports both even and uneven mappings, and offers fast auto-scheduling. However, it overlooks key issues such as data reuse, limited memory capacity, and interactions with off-chip memory.

Similarly, Huang et al. proposed an integer-only quantization scheme and algorithm-hardware co-design for running Transformer-like networks on edge devices [10]. While their approach enhances matrix multiplication and self-attention through optimized memory management, it relies on sequential memory bank access. This limits parallelism, underutilizes memory resources, and struggles with varying workloads and irregular memory access patterns.

Furthermore, another promising direction involves compute-in-memory (CIM) architectures. Jain et al. [11] proposed a CIM-based accelerator that combines distributed memory tiles for efficient multiply–accumulate (MAC) operations with dedicated compute cores for various tasks. A dense 2D mesh supports efficient data exchange and workload pipelining. However, the main challenge remains managing data efficiently within CIM arrays, as data movement and off-chip memory latency still hinder performance, especially in edge scenarios where fast and efficient memory access is essential.

Focusing on the aforementioned challenges: 1) inefficient off-chip memory access and data reuse, 2) suboptimal memory management, and 3) inadequate data scheduling for computing, our proposed architecture addresses these by combining a multi-path rolling data refresh mechanism with layer-wise bank allocation. This approach improves data transfer efficiency, lowers latency, and optimizes memory utilization, thereby improving energy efficiency in AI accelerators.

III. Proposed Architecture

Fig. 1 illustrates the proposed scalable multi-path rolling data refresh and layer-wise bank allocation architecture for edge-AI accelerators. The architecture is designed to optimize memory access and data reuse by employing five rolling-refresh direct memory accesses (RRDMAs), along with a bank allocation strategy to enhance on-chip data management.

The external memory (DDR) is accessed through an AMBA AXI-based interface, which supports burst transfers. Data are fetched using two external DMAs: external activation DMA (XADMA) and external weight DMA (XWDMA), which manage activation and weight transfers from the external memory to internal memory, respectively. These DMAs implement rolling-based data refresh, ensuring that memory banks are updated sequentially in a circular fashion rather than simultaneously, allowing efficient single-port SRAM deployment. The rolling refresh mechanism updates one bank per configurable cycles, avoiding conflicts between read and write operations in the same bank while maximizing memory bandwidth efficiency. In particular, the rolling-based data refresh

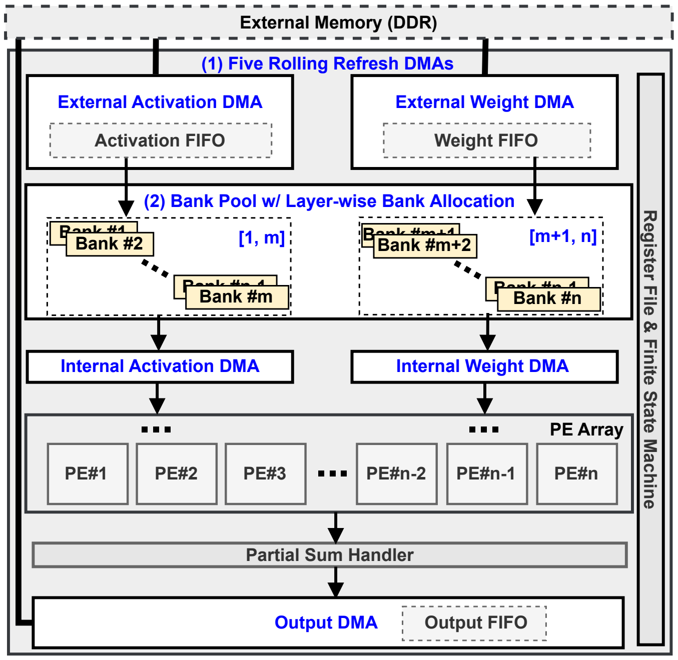


Fig. 1: Proposed architecture for scalable external memory access and on-chip storage.

mechanism demonstrates compatibility with CIM and near-memory computing (NMC) architectures. This synergy arises from the inherent design characteristics of modern memory-processor integrations, where PEs are either directly interfaced with memory modules or physically embedded within memory banks. When implemented in multi-bank memory architectures, this mechanism optimizes three critical aspects of system performance: 1) efficient utilization of off-chip memory bandwidth, 2) maximized throughput across all memory banks, and 3) full exploitation of parallel computing through coordinated PE array. The combined effect significantly enhances data reuse while maintaining balanced resource utilization.

The layer-wise bank allocation strategy (Fig. 1, middle) dynamically assigns on-chip memory banks to store weights and activations across different NN layers. The total number of memory banks is n, with banks 1 to m ($< n$) allocated for activations and banks $m+1$ to n allocated for weights. Each layer is assigned a dedicated set of banks to prevent resource contention and underutilization, as elaborated in Section III-B. Two internal DMAs (internal activation DMA (IADMA) and internal weight DMA (IWDMA)) efficiently fetch activations and weights from their respective allocated banks and stream them into the PE array for computation. The PE array then performs parallel operations for AI tasks. In addition, most AI accelerators integrate partial sum handler (PSH) to handle temporary results in buffers and accumulate them until the final results are obtained. When the current round of calculations is completed, the PSH feeds the final result into the output DMA (ODMA), which also follows a rolling-refresh style similar to other DMAs. The ODMA sequentially transfers final results from buffers to external memory, ensuring continuous data movement and high computational utilization. A finite

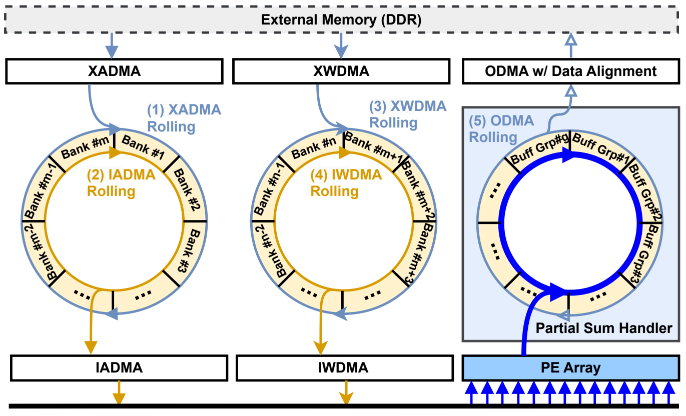


Fig. 2: Data paths and rolling data refresh strategy.

state machine (FSM) and register file orchestrate all these operations, ensuring seamless synchronization of data flow.

A. Multi-Path Rolling Data Refresh Mechanism

The proposed architecture employs a rolling data refresh mechanism with five independent paths to ensure efficient memory access and computational throughput, as shown in Fig. 2. These five paths are directly tied to the data load and store operations, while the blue path from the PE array to the output buffer through the PSH is fixed. This configuration is designed to improve memory access efficiency by aligning each path with specific load/store cycles and computation tasks, ensuring a balanced data flow and high throughput. The external memory serves as the primary data source. In addition, to match the bandwidth resources of the ODMA read channel, XWDMA and XADMA actually share a write channel. That is, only one of them will be activated in each burst transmission. These two DMAs (2 rolling paths) manage the data movement from DDR to on-chip memory through a rolling strategy, ensuring that each memory bank receives data sequentially in a queue-like manner. For example, during the first cycle, Bank #1 is updated, followed by Bank #2 in the next cycle, and so on, until all activation or weight memory banks have been refreshed. In addition, the storage of weight banks (e.g., in convolution operation) is allocated according to the number of kernels. Assume there are k (i.e., $n-m$) weight banks and $3\times k$ kernels. Then kernels (#1, #$k+1$, #$2k+1$) are stored in bank #$m+1$, kernels (#2, #$k+2$, #$2k+2$) are stored in bank #$m+2$, and so on. If there is not enough storage, it will be allocated to the next round of storage and continue calculation. This cyclic refresh mechanism eliminates the need for bulk updates, allowing a continuous data flow and balanced bandwidth utilization. Similarly, IADMA and IWDMA introduce two additional rolling paths, ensuring that activations and weights are sequentially fetched from memory banks and fed into the PE array for computation. The FSM governing these operations guarantees that read and write operations are performed on different banks in each cycle, preventing conflicts that would arise if both read and write operations targeted the same bank simultaneously. Unlike conventional methods that access a single bank for multiple cycles before switching to the next, which usually require dual-port or two-port SRAMs, the proposed rolling refresh strategy staggers memory access across different banks and time slots. This scheduling naturally avoids port conflicts between DMA units, allowing efficient multiplexing of SRAM ports without performance degradation. As a result, all on-chip memory banks can use single-port SRAMs, significantly reducing area overhead while maintaining high computational throughput.

Furthermore, the PSH maintains a structured storage scheme using multiple buffer groups. As a computation round nears completion, a rolling strategy enables the sequential output of each group's results in a queued manner (i.e., one group is fully output before the next begins) ensuring an orderly and continuous data stream. At peak throughput, these groups can be output without interruption. Subsequently, the ODMA then transfers the results from group #1 to #g to external memory in a rolling queue approach, cyclically repeating the cycle until the current layer's computation is complete. This approach improves memory bandwidth efficiency and ensures that results are promptly stored for the next processing stage.

B. Layer-Wise Bank Allocation and Data Reuse Strategy

A layer-wise bank allocation strategy is proposed to minimize external memory access and maximize on-chip data reuse. As specified in Algorithm 1, an example of convolution operation is offered. In addition, since convolution operations and matrix multiplication can be directly converted to each other, this method is also applicable to matrix operations [12].

The algorithm determines the best partitioning of memory banks for input activations and weights while ensuring efficient spatial and channel-wise tiling strategies. It begins by initializing hardware and convolutional layer parameters. The key hardware constraints include the number of available memory banks N_{bank}, the PE array size PE_n, and the number of MACs per PE PE_m. The bit width of calculation precision W_{data} and the depth D_{bank} and the width W_{bank} of each memory bank are also taken into consideration. The convolutional parameters include the input activation dimensions (H_{in}, W_{in}), the number of input and output channels (C_{in}, C_{out}), kernel size (K_x, K_y), stride size (S_x, S_y), and padding size (P_x, P_y).

Once the parameters are set, the algorithm calculates the output activation dimensions H_{out} and W_{out} based on the given parameters. Next, it computes the minimum required memory banks for storing activations and weights, ensuring that their total does not exceed the available memory banks N_{bank}. The minimum number of memory banks and memory capability are determined to ensure that at least W_{out} results per PE can be output in one calculation round in current layer.

To find the optimal memory allocation, the algorithm iterates over possible memory bank allocations, determining that assigning two subsets of memory banks to weights and activations results in the least external memory access. For each allocation, it calculates the channel partitioning strategy by determining the number of output channels that can be processed simultaneously C_{slice}, ensuring that the

Algorithm 1 Search for optimal bank allocation and data reuse strategy to minimize external memory access and maximize data reuse for NN mapping

```
Require: Convolution Layer Parameters:
        $H_{in}, W_{in}$: Input activation dimensions
        $C_{in}, C_{out}$: Input/output channel count
        $K_x, K_y, S_x, S_y, P_x, P_y$ : Kernel & Stride & Padding size
    Hardware Parameters:
        $PE_m$: Number of MACs per PE
        $PE_n$: Number of PEs in the PE array
        $N_{bank}$: Number of memory banks
        $W_{data}$: Data bit width
        $D_{bank}$: Depth of each memory bank
        $W_{bank}$: $PE_m \times W_{data}$ {Bank width in bits}
Ensure: Optimal memory bank allocation and tiling parameters
 1: Compute Output Activation Dimensions and Overlap Area:
 2: $H_{out} \leftarrow \lfloor (H_{in} + 2P_y - K_y)/S_y \rfloor + 1$
 3: $W_{out} \leftarrow \lfloor (W_{in} + 2P_x - K_x)/S_x \rfloor + 1$
 4: $\Delta H \leftarrow K_y - S_y$
 5: Calculate Minimum Storage Requirements:
 6: $D_{per\_row} \leftarrow W_{in} \times \lceil C_{in}/PE_m \rceil$ {Elements per row}
 7: $N_{act}^{min} \leftarrow \lceil D_{per\_row} \times K_y / D_{bank} \rceil$ {Minimum required memory
    banks for input activation}
 8: $N_{wt}^{min} \leftarrow \lceil (K_x \times K_y \times PE_n \times \lceil C_{in}/PE_m \rceil)/D_{bank} \rceil$ {Minimum
    required memory banks for weights}
 9: if $N_{act}^{min} + N_{wt}^{min} > N_{bank}$ then
10:     Return Error: Insufficient memory resources
11: end if
12: Iterate Over Memory Allocation Configurations:
13: Initialize $MA_{current} = \infty$
14: for $N_{act} = N_{act}^{min}$ to $N_{bank} - N_{wt}^{min}$ do
15:     $N_{wt} \leftarrow N_{bank} - N_{act}$
16:     $C_{slice} \leftarrow \min\left(\left\lfloor \frac{D_{bank} \times N_{wt}}{K_x \times K_y \times PE_n \times \lceil C_{in}/PE_m \rceil} \right\rfloor \times PE_n, C_{out}\right)$
17:     Channel Partitioning Strategy:
18:     $C_{split} \leftarrow \lceil C_{out}/C_{slice} \rceil$ {Output channel split count}
19:     $C_{last} \leftarrow \begin{cases} C_{slice} & C_{out} \mod C_{slice} = 0 \\ C_{out} \mod C_{slice} & \text{otherwise} \end{cases}$
20:     Spatial Tiling Strategy:
21:     $H_{first}^{out} \leftarrow \lfloor (D_{bank} \times N_{act}/D_{per\_row} + P_y - K_y)/S_y \rfloor + 1$
22:     $H_{first}^{in} \leftarrow (H_{first}^{out} - 1) \times S_y + K_y - P_y$ {First block input height}
23:     $H_{mid}^{out} \leftarrow \lfloor (D_{bank} \times N_{act}/D_{per\_row} - K_y)/S_y \rfloor + 1$
24:     $H_{mid}^{in} \leftarrow (H_{mid}^{out} - 1) \times S_y + K_y$ {Middle block input height}
25:     if $H_{first}^{out} \geq H_{out}$ then
26:         $H_{first}^{out} \leftarrow H_{out},\ H_{first}^{in} \leftarrow H_{in}$
27:     end if
28:     $R_{remain} \leftarrow H_{out} - H_{first}^{out}$
29:     $H_{split} \leftarrow \lceil R_{remain}/H_{mid}^{out} \rceil + 1$
30:     $H_{last}^{out} \leftarrow \begin{cases} H_{mid}^{out} & R_{remain} \mod H_{mid}^{out} = 0 \\ R_{remain} \mod H_{mid}^{out} & \text{otherwise} \end{cases}$
31:     if $H_{split} \geq 2$ then
32:         $H_{last}^{in} \leftarrow H_{in} - H_{first}^{in} + \Delta H - (H_{split} - 2) \times (H_{mid}^{in} - \Delta H)$
            {Last block input height}
33:     end if
34:     Memory Access Estimation of Weight/Activation Reuse:
35:     $MA_{wt}/PE_m \leftarrow K_x \times K_y \times C_{out} \times \left(\lceil \frac{C_{in}}{PE_m} \rceil\right) + \left(D_{per\_row} \times H_{in} + D_{per\_row} \times \Delta H \times (H_{split} - 1)\right) \times C_{split}$
36:     $MA_{act}/PE_m \leftarrow H_{split} \times K_x \times K_y \times C_{out} \times \left(\lceil \frac{C_{in}}{PE_m} \rceil\right) + D_{per\_row} \times H_{in} + D_{per\_row} \times \Delta H \times (H_{split} - 1)$
37:     Select Optimal Configuration:
38:     if $MA_{wt} < MA_{current}$ then
39:         $MA_{current} \leftarrow MA_{wt}$; Store weight reuse $Method$
40:     else if $MA_{act} < MA_{current}$ then
41:         $MA_{current} \leftarrow MA_{act}$; Store activation reuse $Method$
42:     end if
43: end for
44: Return Final Output Configurations for NN Deployment:
45: $N_{act}^{opt}, N_{wt}^{opt}$: Optimal memory bank allocation
46: $Method^{opt} \in \{WeightReuse, ActivationReuse\}$
47: All optimal parameters relevant in the search process
```

Note: All related parameters are stored in the register file, enabling the FSM to dynamically execute corresponding computations in the AI accelerator.

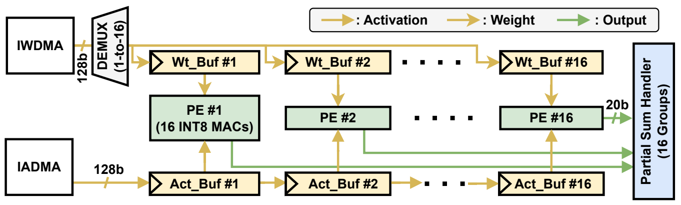


Fig. 3: Design of PE array for evaluation.

data fit within the allocated memory banks. The number of required partitions C_{split} and the remaining channels in the last partition C_{last} are computed accordingly. Similarly, the spatial tiling strategy is determined by partitioning the output activation height into multiple segments, ensuring efficient memory utilization. The output tile height $H_{first/mid/last}^{out}$, the input tile height $H_{first/mid/last}^{in}$, and the number of partitions H_{split} are derived based on the available memory space.

After defining the tiling strategies, the algorithm estimates the total memory access consumption under two different strategies, including 1) weight reuse strategy prioritizes keeping weights in memory while reloading activations as needed, 2) activation reuse strategy prioritizes keeping activations in memory while reloading weights. Since the weights or activations calculated in the current round may be reused in the next round, in addition to properly allocating memory banks, choosing a reasonable data reuse scheme can further reduce external memory access and increase on-chip data reuse.

The external memory access of weight reuse MA_{wt} and activation reuse MA_{act} is computed by considering the number of channel and spatial partitions, and the PE array size. The algorithm then selects the configuration with the lowest total memory access. Finally, it outputs the optimal memory allocation configuration, including the number of allocated banks for activations N_{act}^{opt} and weights N_{wt}^{opt}, relevant parameters for the configuration of AI accelerator, and the chosen optimization strategy $Method^{opt}$. This approach ensures that the memory footprint is minimized, reducing off-chip memory access while maximizing computational throughput and on-chip data reuse.

IV. An Edge-AI Accelerator for Evaluation

A. Design of Processing Element Array

To evaluate the proposed architecture, a typical PE array is constructed as shown in Fig. 3. The array consists of 16 PEs (PE_n), each containing 16 INT8 MAC units (PE_m). Each PE outputs the accumulated result of 16 INT8×INT8 operations. Activations and weights (128-bit) are fetched via IADMA and IWDMA into dedicated activation (Act_Buf) and weight (Wt_Buf) buffers. The PEs generate 20-bit partial sums, which are stored in a partial sum holder (PSH) for further accumulation. The PSH consists of 16 buffer groups, each storing 16 results, totaling 256 temporary results. These partial sums are accumulated until final results are obtained, which are then truncated to 8-bit to align the data width.

Furthermore, to build an accelerator for comprehensive evaluation, a total of 16 (N_{bank}) memory banks are implemented corresponding to the proposed architecture. The size of each

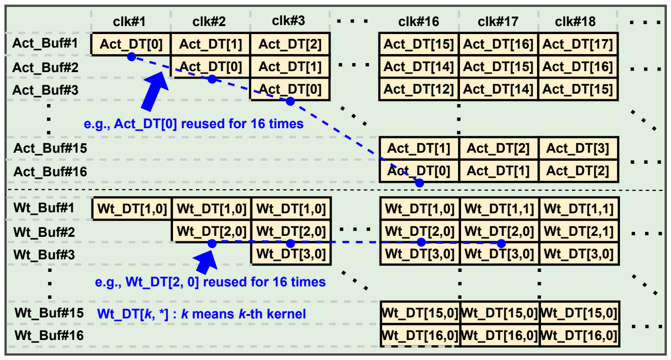


Fig. 4: Data update strategy in PE array.

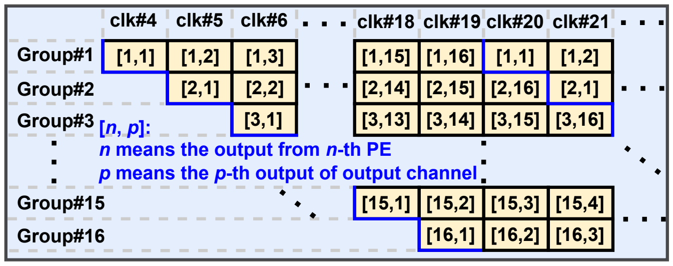


Fig. 5: Data update strategy in PSH for accumulation.

bank is 128b(W_{bank})×2048(D_{bank}), a total of 512KB in the accelerator. The memory size is determined according to the size of prevalent NN models and the number of on-chip PEs. Moreover, these 16 memory banks can be reasonably allocated according to Algorithm 1 in Section III-B to achieve the most efficient external memory access and on-chip data reuse.

B. Data Update Strategy Related to PE Array

Fig. 4 illustrates the data update strategy in the PE array, showing how activations and weights interact to achieve efficient data reuse. IWDMA is responsible for fetching weight data from the bank memory and distributing it to multiple weight buffers (Wt_Buf #1 to #16), each corresponding to a PE. At peak throughput case, each Wt_Buf updates data every 16 cycles, which means that each weight can be reused 16 times. Similarly, IADMA loads activation data into Act_Buf #1 and passes the data to Act_Buf #16 in a pipelined manner. The 16-stage pipeline means that each activation can be reused 16 times. The temporary results (partial sums) generated by PEs are forwarded to the PSH, which can accumulate partial sums before writing the final results to external memory. The PE array structure ensures efficient data reuse because the weights are kept in buffers as different activations pass through the PE, minimizing the memory bandwidth requirements.

Fig. 5 details the data update strategy within the PSH, highlighting how partial sums are accumulated across clock cycles. Each group represents a set of partial sums generated by different PEs, which are progressively updated over time. Initially, at clk#4, PE#1 outputs an initial partial sum [1,1]. As the clock progresses, each PE produces additional results, with subsequent activations and weights contributing to further accumulations. By clk#19, each group has computed multiple

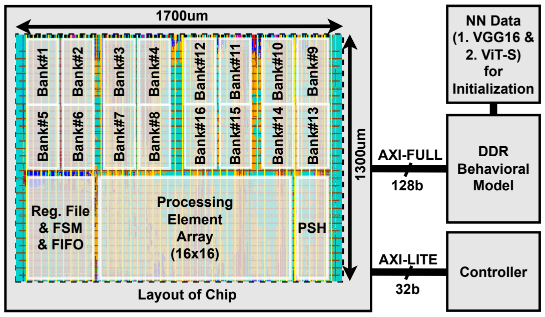


Fig. 6: Experimental setup for performance evaluation.

intermediate partial sums. At clk#20, the buffer starts processing a new set of accumulations starting from [1,1] while still retaining previous results, allowing efficient accumulation. This approach ensures that the output remains consistent across multiple cycles. The strategy effectively balances data reuse and pipeline execution by maintaining intermediate sums within the PSH, thereby reducing memory access overhead.

V. Experimental Results

A. Experimental Setup

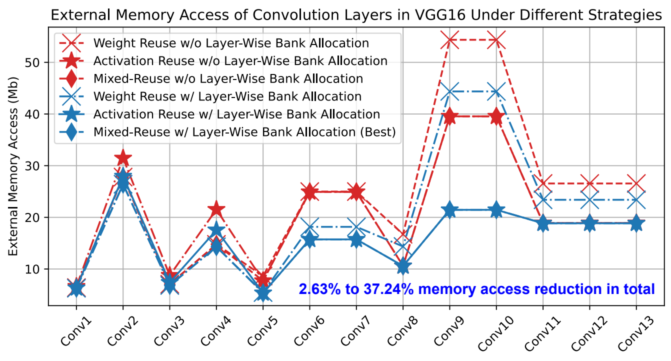
The experimental setup is shown in Fig. 6. The proposed design is implemented on a commercial 28nm CMOS process at 0.8V, using standard digital IC design flow with commercial EDA tools (i.e., Synopsys Design Complier for logic synthesis and Cadence Innovus for physical design). The post-layout simulation is performed in Synopsys VCS with extracted parasitic resistance and capacitance from the layout. Two models, VGG16 and ViT-Small, are used to evaluate the performance. During the initialization process, the testbench loads the model data into the DDR behavioral model, and then the controller configures the register file in the chip through the AXI-Lite bus to activate the accelerator. The accelerator exchanges data with DDR through the AXI-Full bus.

B. Evaluation of Rolling Data Refresh

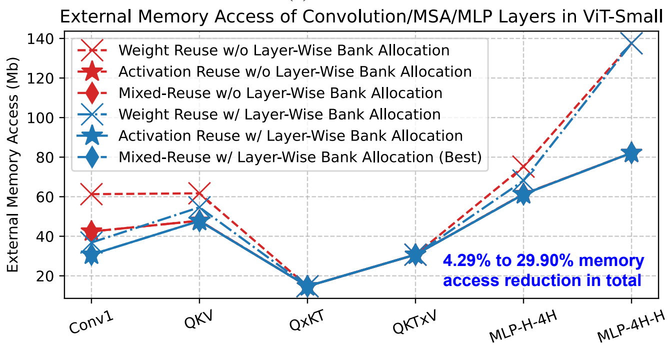
The rolling data refresh strategy improves memory access efficiency by distributing refresh operations over time, reducing memory stalls and ensuring a more balanced workload. Another key advantage is that it schedules read and write operations to avoid conflicts, allowing to use single-port SRAM instead of two-port SRAM. This significantly reduces area overhead, as single-port SRAM requires fewer transistors per bit. For a 128b×2048 SRAM, single-port block occupies 0.061 mm^2, while two-port block takes 0.122 mm^2 (50% area reduction). This optimization leads to a reduction of 30.6% in the total area of the chip compared with an identical design realized by two-port SRAMs, significantly improving the area efficiency. Additionally, it lowers power consumption and simplifies memory design by reducing routing complexity.

C. Evaluation of Layer-Wise Bank Allocation and Data Reuse

Fig. 7 demonstrates the effectiveness of our layer-wise bank allocation and data reuse strategy in reducing external memory access across different NN architectures. Different



(a) VGG16



(b) ViT-Small

Fig. 7: External memory access of related layers in NNs.

NN architectures exhibit varying memory demands: models with deeper channels tend to have larger activation sizes, while models with larger kernels require more weight storage. Given these variations, the bank allocation strategy must be designed to accommodate a wide range of models rather than being optimized for a single case. To establish a fair baseline comparison, we assume an even distribution of the 16 banks for weights and activations without layer-wise bank allocation. This strategy ensures that the memory allocation is balanced across different models. Conversely, allocating too few banks for weights or activations can hinder large-kernel models and deeper networks, causing computation failures or memory bottlenecks. Thus, VGG16 uses 8 banks for weight and activation respectively. However, ViT-Small uses 6 banks for weight and 10 banks for activation, otherwise it cannot complete the calculation due to insufficient memory. Compared to traditional approaches without bank allocation or mixed-data reuse, our strategy achieves 2.63% to 37.24% reduction in memory access for VGG16 and 4.29% to 29.90% reduction for ViT-Small. This improvement is attributed to the optimized memory partitioning, which assigns dedicated banks for activations and weights, minimizing data transfers. Additionally, the data reuse mechanism efficiently leverages on-chip buffers, further reducing the reliance on costly external memory access, enhancing overall performance. On the other hand, for some layers with a few parameters and computational overhead (e.g., Q$\times$KT), the on-chip memory can store all or most of the data at once, so the optimization is not significant.

D. *Comparison with Existing AI Accelerators*

TABLE I: Performance comparison with SOTA designs.

	This Work	**[13] ISSCC'23**	**[14] ISSCC'23**	**[15] VLSI'23**
Technology	28nm	28nm	22nm FDX	40nm
Evaluation Method	Post-layout Simulation	Chip Measurement	Chip Measurement	Chip Measurement
Architecture	Digital Accel.	Digital Accel.	RISC-V + RBE Accel.	Digital Accel.
Supply [V]	0.8	0.66-1.3	0.5-0.8	0.7-0.9
Freq. [MHz]	700	100-500	420	5-80
Area [mm^2]	2.21	7.81	18.7	4.92
Power [mW]	34.85-42.70	17-174	12.8-123	0.793-1.032
Memory [kB]	512	1120	1152	206
Precision	INT8	INT8	INT2-8 (Accel.)	A:8b; W:1-8b
Area Efficiency [TOPS/mm^2]	0.162	0.131	0.034@RBE2x2	0.004@A8b,W1b
[e]Peak Power Efficiency [TOPS/W]	10.28@INT8[1] 8.81@VGG16[2] 7.36@ViT-S[3]	7.28@ResNet50 7.15@MobileNetV1 5.10@MobileNetV2	3.81@RBE2x2 1.96@ResNet20 1.79@ResNet18	30.41@A8b,W1b 1.37@MobileNetV1

[1] 50% sparsity, 100% PE utilization
[2] 19.33% / 72.55% sparsity for weight / activation in VGG16, accuracy: 71.68%
[3] 12.78% / 55.14% sparsity for weight / activation in ViT-Small, accuracy: 76.82%
[e] Power efficiency scaled to [0.8V 28nm] = Energy efficiency$\times \frac{process}{28nm} \times (\frac{Voltage}{0.80V})^2$

In Table I, we compare the accelerator built on the proposed architecture with other SOTA designs. As seen from the comparison with [13]–[15] in real end-to-end NN evaluations, our design achieves a power efficiency improvement ranging from 1$\times$ to 6.4$\times$. This improvement is largely due to the efficient layer-wise bank allocation and data reuse strategy that aligns seamlessly with our proposed architecture. Furthermore, the rolling data refresh strategy contributes to a significant improvement in area efficiency, ranging from 1.2$\times$ to 40.5$\times$.

VI. Conclusion

This paper presents a scalable architecture that addresses key challenges in resource-constrained AI accelerators for edge computing, specifically focusing on optimizing memory access and management. By integrating a multi-path rolling data refresh mechanism and layer-wise bank allocation, the proposed design enhances the efficiency of off-chip and on-chip data interactions, significantly reducing latency and minimizing memory overhead. The layer-wise allocation of memory banks optimizes memory access based on the structure of specific NNs, leading to improved memory efficiency and reduced latency. A case study on a 28nm AI accelerator demonstrates impressive power efficiency of 10.28 TOPS/W and a 2.63% to 37.24% reduction in external memory access for VGG16 and ViT-Small models. The proposed architecture shows promising potential for enhancing the performance and efficiency of AI accelerators in edge computing environments.

Acknowledgment

This work is partially supported by the Grant-in-Aid for Scientific Research (S) from Japan Society for the Promotion of Science (JSPS) under Grant 24H00073, National Natural Science Foundation of China under Grant No. 62274081, and Grant no. 2023QN10X177. Dr. Xiong's contribution to this work is limited to his interactions with Drs. Cheng and Hashimoto at Kyoto University, Kyoto, Japan.

REFERENCES

[1] T. Mohaidat and K. Khalil, “A Survey on Neural Network Hardware Accelerators,” in IEEE Transactions on Artificial Intelligence, vol. 5, no. 8, pp. 3801-3822, Aug. 2024, doi: 10.1109/TAI.2024.3377147.

[2] Z. Chang, S. Liu, X. Xiong, Z. Cai and G. Tu, “A Survey of Recent Advances in Edge-Computing-Powered Artificial Intelligence of Things,” in IEEE Internet of Things Journal, vol. 8, no. 18, pp. 13849-13875, 15 Sept.15, 2021, doi: 10.1109/JIOT.2021.3088875.

[3] N. Zhang, S. Ni, L. Chen, T. Wang and H. Chen, “High-Throughput and Energy-Efficient FPGA-Based Accelerator for All Adder Neural Networks,” in IEEE Internet of Things Journal, doi: 10.1109/JIOT.2025.3543213.

[4] T. Sipola, J. Alatalo, T. Kokkonen and M. Rantonen, “Artificial Intelligence in the IoT Era: A Review of Edge AI Hardware and Software,” 2022 31st Conference of Open Innovations Association (FRUCT), Helsinki, Finland, 2022, pp. 320-331, doi: 10.23919/FRUCT54823.2022.9770931.

[5] D. Kudithipudi et al., “Design principles for lifelong learning AI accelerators.” Nature Electronics 6.11 (2023): 807-822.

[6] A. Feldmann, C. Golden, Y. Yang, J. S. Emer and D. Sanchez, “Azul: An Accelerator for Sparse Iterative Solvers Leveraging Distributed On-Chip Memory,” 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO), Austin, TX, USA, 2024, pp. 643-656, doi: 10.1109/MICRO61859.2024.00054.

[7] C. Yi, S. Jian, Y. Tan and Y. Zhang, “HMO: Host Memory Optimization for Model Inference Acceleration on Edge Devices,” 2024 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Kuching, Malaysia, 2024, pp. 2813-2819, doi: 10.1109/SMC54092.2024.10831215.

[8] A. Symons, L. Mei and M. Verhelst, “LOMA: Fast Auto-Scheduling on DNN Accelerators through Loop-Order-based Memory Allocation,” 2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS), Washington DC, DC, USA, 2021, pp. 1-4, doi: 10.1109/AICAS51828.2021.9458493.

[9] Q. Cheng et al., “A 13-34 TOPS/W Edge-AI Processor Featuring Booth-Value-Confined Accelerator, Near-Memory Computing, and Contiguity-Aware Mapping,” 2024 IEEE Asian Solid-State Circuits Conference (A-SSCC), Hiroshima, Japan, 2024, pp. 1-3, doi: 10.1109/A-SSCC60305.2024.10849341.

[10] M. Huang, J. Luo, C. Ding, Z. Wei, S. Huang and H. Yu, “An Integer-Only and Group-Vector Systolic Accelerator for Efficiently Mapping Vision Transformer on Edge,” in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 70, no. 12, pp. 5289-5301, Dec. 2023, doi: 10.1109/TCSI.2023.3312775.

[11] S. Jain et al., “A Heterogeneous and Programmable Compute-In-Memory Accelerator Architecture for Analog-AI Using Dense 2-D Mesh,” in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 31, no. 1, pp. 114-127, Jan. 2023, doi: 10.1109/TVLSI.2022.3221390.

[12] A. Anderson, A. Vasudevan, C. Keane and D. Gregg, “High-Performance Low-Memory Lowering: GEMM-based Algorithms for DNN Convolution,” 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Porto, Portugal, 2020, pp. 99-106, doi: 10.1109/SBAC-PAD49847.2020.00024.

[13] C. -Y. Du et al., “A 28nm 11.2TOPS/W Hardware-Utilization-Aware Neural-Network Accelerator with Dynamic Dataflow,” 2023 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, USA, 2023, pp. 1-3, doi: 10.1109/ISSCC42615.2023.10067774.

[14] F. Conti et al., “22.1 A 12.4TOPS/W @ 136GOPS AI-IoT System-on-Chip with 16 RISC-V, 2-to-8b Precision-Scalable DNN Acceleration and 30%-Boost Adaptive Body Biasing,” 2023 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, USA, 2023, pp. 21-23, doi: 10.1109/ISSCC42615.2023.10067643.

[15] J. Suzuki et al., “Pianissimo: A Sub-mW Class DNN Accelerator with Progressive Bit-by-Bit Datapath Architecture for Adaptive Inference at Edge,” 2023 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits), Kyoto, Japan, 2023, pp. 1-2, doi: 10.23919/VLSITechnologyandCir57934.2023.10185293.