MAICC : A Lightweight Many-core Architecture with In-Cache Computing for Multi-DNN Parallel Inference

Renhao Fan
frh21@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Yikai Cui
cuiyk19@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Qilin Chen
cql22@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Mingyu Wang
wangmingyu@mail.sysu.edu.cn
Sun Yat-sen University
Guangzhou, China

Youhui Zhang
zyh02@tsinghua.edu.cn
Tsinghua University
Beijing, China
Zhongguancun Laboratory
Beijing, China

Weimin Zheng
zwm-dcs@tsinghua.edu.cn
Tsinghua University
Beijing, China

Zhaolin Li*
lzl73@tsinghua.edu.cn
Tsinghua University
Beijing, China

ABSTRACT

The growing complexity and diversity of neural networks in the fields of autonomous driving and intelligent robots have facilitated the research of many-core architectures, which can offer sufficient programming flexibility to simultaneously support multi-DNN parallel inference with different network structures and sizes compared to domain-specific architectures. However, due to the tight constraints of area and power consumption, many-core architectures typically use lightweight scalar cores without vector units and are almost unable to meet the high-performance computing needs of multi-DNN parallel inference. To solve the above problem, we design an area- and energy-efficient many-core architecture by integrating large amounts of lightweight processor cores with RV32IMA ISA. The architecture leverages the emerging SRAM-based computing-in-memory technology to implement vector instruction extensions by reusing memory cells in the data cache instead of conventional logic circuits. Thus, the data cache in each core can be reconfigured as the memory part and the computing part with the latter tightly coupled with the core pipeline, enabling parallel execution of the basic RISC-V instructions and the extended multi-cycle vector instructions. Furthermore, a corresponding execution framework is proposed to effectively map DNN models onto the many-core architecture by using intra-layer and inter-layer pipelining, which potentially supports multi-DNN parallel inference. Experimental results show that the proposed MAICC architecture obtains a 4.3× throughput and 31.6× energy efficiency over CPU (Intel i9-13900k). MAICC also achieves a 1.8× energy efficiency over GPU (RTX 4090) with only 4MB on-chip memory and 28 mm^2 area.

ACM Reference Format:
Renhao Fan, Yikai Cui, Qilin Chen, Mingyu Wang, Youhui Zhang, Weimin Zheng, and Zhaolin Li. 2023. MAICC : A Lightweight Many-core Architecture with In-Cache Computing for Multi-DNN Parallel Inference . In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3613424.3614268

*Corresponding Author: Zhaolin Li (lzl73@tsinghua.edu.cn)

This work is licensed under a Creative Commons Attribution International 4.0 License.

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3614268

1 INTRODUCTION

Complex intelligence tasks in real-world scenarios require an increasing number of deep neural network (DNN) models of different structures and sizes to work in collaboration. Taking autonomous driving as an example, various kinds of environmental data are collected by dozens of various sensors, such as cameras, radars, and LiDARs, and are fed into multiple DNNs for real-time perception, cognition, and decision-making [2, 24, 36, 52]. Moreover, the size of DNN models is also rapidly growing at 240 times every two years [22]. These trends pose significant challenges to hardware platforms in terms of throughput, energy consumption, and flexibility.

Many domain-specific NN accelerators have been proposed for high-performance and energy-efficient DNN inference [7, 9, 18, 30, 35, 50]. These NN accelerators employ dataflow on 1D or 2D processing engine (PE) arrays to achieve efficient data reuse. However, customized NN accelarators generally face the problem of programmability and flexibility. The operation types supported by NN accelarators are entirely determined by the hardware circuits of PEs and special function units with limited software programmability. In addition, each NN accelarator requires a separate scheduling and compilation framework, increasing the deployment cost of the model.

On the other hand, traditional multi-core CPUs and GPUs are capable of supporting various operations through software programming. However, different cores in CPUs and GPUs can only communicate through the complex cache hierarchy, which introduces additional power consumption but also makes memory bandwidth the bottleneck. As a result, high-performance CPUs and GPUs consume hundreds of watts [30] and cannot meet the high energy-efficiency requirements of NN inference.

In order to achieve both energy efficiency and flexibility, a promising architecture for multi-model inference is the many-core architecture [4, 5, 14, 15, 38, 54]. Typical, many-core architecture is a 2D array of processors connected by a mesh network-on-chip (NoC). The many-core employs software-programmable processors as nodes to achieve flexibility, while its explicit spatial structure enables direct communication and dataflow between cores, which reduces data movement and achieves energy efficiency.

However, due to limited area and energy budget, many-core can only uses lightweight scalar cores without vector units, resulting in limited performance. Traditional vector computing units,including vector processors and SIMD units, require considerable logic circuits and vector registers to implement computations, which dramatically increase the area and energy consumption. The recent RISC-V vector processors have an area and power consumption of 2.54× and 3.35× that of scalar cores [6, 39], which is unacceptable for many-core architectures.

In order to improve performance under this tight limitation, it is necessary to introduce efficient heterogeneous vector units into the lightweight scalar core. The emerging SRAM-based compute-in-memory (CIM) technology is a promising solution to achieve in-situ low-precision vector computing by reusing memory cells as computing units [1, 17, 19, 43, 51]. Compared with conventional area-consuming vector architectures, this in-SRAM-computing technology only requires adding some peripheral logic with a negligible area overhead around the SRAM array, and the data can be processed in-situ without transferring from memory to vector registers, which helps to improve performance without significant area and energy cost.

In this paper, we propose MAICC, a Many-core Architecture with In-Cache Computing for multi-DNN model inference. We transform the scratchpad in each many-core node into a computing memory (CMem) that supports in-situ fixed-point vector MAC operations and integrate it into the pipeline of RISC-V lightweight core. In this architecture, CMem efficiently performs the compute-intensive matrix multiplication and convolution, while the RISC-V pipeline is responsible for running auxiliary functions, shuffling data, and communicating with other nodes. This tightly-coupled heterogeneous architecture fully combines the advantages of CIM and many-core, providing performance, energy efficiency, and flexibility at the same time. Based on MAICC, we propose a complete execution framework that fully utilizes intra-layer streaming and inter-layer pipelining to spatially map DNN models onto the many-core architecture, thus reducing latency and data movement.

We conduct detailed experiments to evaluate MAICC at the single-core and the many-core level. Experimental results show that compared with Neural Cache [17], an existing in-cache computing work, MAICC achieves 2.3× performance and slightly better energy efficiency in performing convolutions on a single core. Compared with CPU (Intel i9-13900k) and GPU (RTX 4090), the proposed 210-core MAICC achieves a 31.6× and 1.8× energy efficiency enhancement with only 28 mm^2 area, respectively.

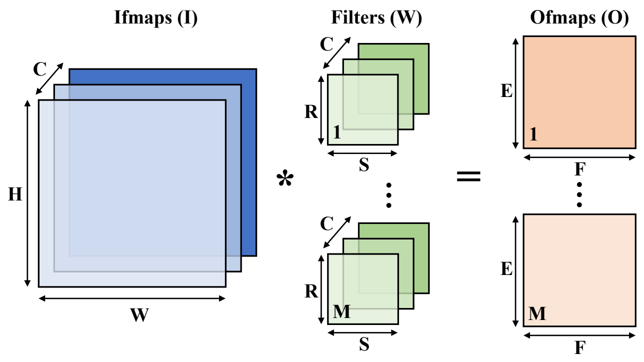

Figure 1: Computation of convolutional layers. The 3D ifmap has C channels of $H*W$ fmaps. The 4D weight tensor includes M filters of size $R*S*C$. Each filter performs convolution with the ifmap on all channels and the results are accumulated into one ofmap. M filters generates M channels of the ofmap.

In summary, the contributions of this paper are as follows:

- **The computing memory.** We propose a novel computing memory structure that not only caches data but also efficiently supports fixed-point vector MAC operations.
- **Intra-core architecture that tightly couples the RISC-V pipeline and the CMem.** We integrate CMem into the RISC-V pipeline and design the multi-cycle instruction set (ISA) extension. The static and dynamic scheduling mechanisms are implemented to enable the parallelism between the pipeline and the CMem.
- **A DNN execution framework for many-core architecture.** We design a complete execution framework to spatially map subsequent layers of DNN onto the proposed many-core architecture, which potentially supports multi-DNN parallel inference.
- **Thorough evaluation of MAICC.** We conduct detailed experiments to evaluate MAICC on ResNet18 [25]. Experimental results show that the 210-core MAICC has better energy efficiency than CPU (Intel i9-13900k) and GPU (RTX 4090).

2 BACKGROUND AND MOTIVATION

2.1 Deep Neural Networks

Convolutional Neural Networks (CNNs) [23, 25, 40, 41, 44, 47], Recurrent Neural Networks (RNNs) [12, 26], and Transformers [16, 49] are commonly used DNN structures. Typically, the DNN structure is a directed acyclic graph (DAG) composed of multiple layers.

The most commonly used layers are fully connected (FC) layers and convolutional (CONV) layers. The FC layer obtains the output vector by multiplying the weight matrix with the input vector. The CONV layer applies a set of filters to the feature maps (fmaps), as shown in Figure 1. The computation of CONV layers is defined as

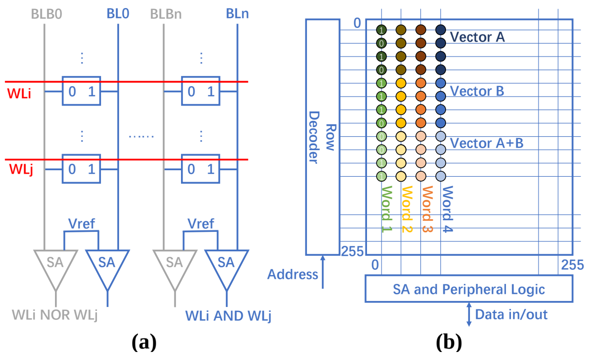

Figure 2: (a) Bit-line computing circuits. Two word-lines WL_i and WL_j are activated simultaneously, and the sense amplifier(SA) reads out the *AND* and *NOR* of two bits from the bit-line (BL) and bit-line bar (BLB). (b) A standard 8KB SRAM array with in-SRAM bit-serial computing [1].

$$O[m][x][y] = b[m] + \sum_{c=0}^{C-1} \sum_{i=0}^{R-1} \sum_{j=0}^{S-1} I[c][x+i][y+j] * W[m][c][i][j]$$

where I and O are 3D input feature maps (ifmap) and output feature maps (ofmap), W is the 4D filter, and b is the 1D bias. The convolution can be decomposed into vector MAC operations (or inner products) at the $R/S/C$ levels, and the matrix multiplication in the FC layer can also be decomposed into vector MACs. Since the FC and CONV layers dominate the computation amount in DNN, optimizing vector MAC operations is a primary consideration in designing new architectures.

Between two adjacent layers are auxiliary function layers, such as activation, pooling, batch normalization, and quantization. The activation layer applies non-linear functions, such as Sigmoid and ReLU, to the output tensor. Maximum or average pooling aggregates adjacent data in fmaps to reduce fmap size. Batch normalization is essentially a linear transformation to normalize fmap distribution. The quantization layer reduces tensor precision from 32-bit floating point to 16/8/4-bit fixed point at the inference stage, which can reduce calculation amount without significant accuracy loss. Different auxiliary functions have various implementations with diverse and irregular computations, which are difficult to implement through hardware circuits solely. Therefore, a general-purpose computing platform should be programmable and able to dynamically configure auxiliary functions according to the demands of different models.

More complex layers appear in advanced DNN structures, such as long short-term memory(LSTM) cells in LSTM and encoders/decoders in Transformer. But they are essentially composed of fully connected layers and the auxiliary functions described above.

2.2 In-SRAM Computing

SRAM array is a two-dimensional array organized by SRAM bit cells, each storing a single bit. These bit cells are interconnected by horizontal word-lines and vertical bit-lines. A standard 8 KB SRAM array includes 256 rows and 256 columns, as shown in Figure 2(b). When an access request occurs, the row decoder activates one word-line, and the bits stored in this row will flow to the bottom along the bit-lines. Then, the requested data will be fetched through the amplifier and column decoder.

In-SRAM computing essentially utilizes the bit-line computing technique proposed by [28, 31]. As shown in Figure 2(a), this technique reports that when multiple word-lines are activated simultaneously, the shared bit-lines can produce *AND* and *NOR* values of the two activate rows. Usually, a subsequent write operation will save the results back into SRAM, achieving in-place computation. Data corruption in the multi-row access is avoided by lowering word-line voltage to bias against write of the SRAM array. With bit-line computing, logic operations of two words can be achieved in-situ inside the SRAM array [1].

In order to support fixed-point arithmetic computation, BLADE [43] adds carry logic after the amplifier and achieves the addition of two words. This way of calculating two complete words at a time is called bit-parallel computing. On the other hand, Neural Cache [17] proposes bit-serial computing, where the data are stored in a transposed way. As shown in Figure 2(b), different bits of a word are located on the same bit-line, and the same bit positions of different words in a vector are placed on the same word-line. In this way, the SRAM array can perform the addition or multiplication on the same bit position of all words in two vectors. Two vectors of 256 words can finish addition in $n + 1$ cycles and multiplication in $n^2 + 5n - 2$ cycles, where n is the bit width of the word. Compared with bit-parallel computing, bit-serial computing simplifies peripheral logic and thus reduces area overhead and operation delay by eliminating carry logic between different bit-lines. Moreover, when dealing with low-precision fixed-point calculation of neural network inference, the bit width n is usually a small value of 8,4 or even 2, which brings high throughput.

Neural Cache further performs a whole DNN model in last-level cache based on the bit-serial vector-level primitives. It first stores an $R * S * C$ filter and an ifmap window of the same size in the SRAM array. Because the filter slides only in the width and height dimension but not in the channel dimension, vectors are organized along the channel dimension. In this way, the filter and ifmap vectors are always aligned. When data are ready, MAC operations are performed through fixed-point vector multiplication and addition primitives. Finally, the partial sums of all C channels are accumulated into a single number by reduction operation, which is implemented by iterative addition and shift. Besides convolution, Neural Cache implements linear layer, pooling, and ReLU in the cache. It also supports quantization and batch normalization with the assistance of CPU.

2.3 Limitations of In-Cache Computing

Previous works of in-SRAM computing are mainly implemented in the cache hierarchy of traditional multi-core CPUs. The cache is hierarchically organized by a large number of SRAM arrays and reaches high parallelism. However, computing in cache hierarchy also faces some shortcomings.

Firstly, the cache works as a co-processor and its control flow is centralized. In Compute Cache[1], Neural Cache[17], and

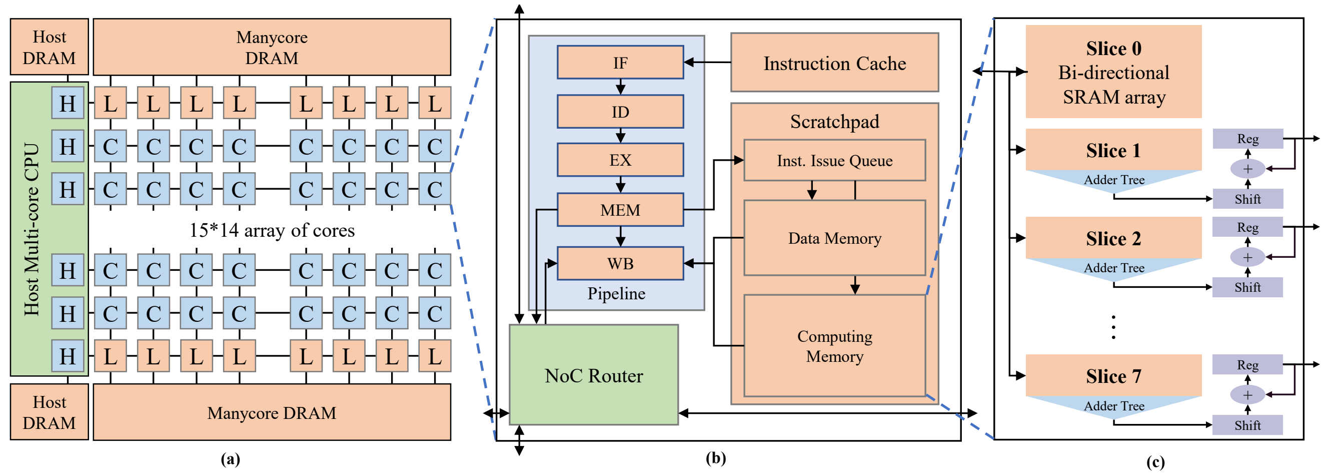

Figure 3: (a) The MAICC architecture is a 16*16 array connected by 2D mesh NoC, which includes a multi-core host CPU (H), 15*14 computing cores (C), and two rows of last-level caches (L). (b) Each core is composed of a lightweight RISC-V pipeline, a 4 KB instruction cache, a 4 KB data memory, a 16 KB CMem, and an NoC router. (c) The CMem consists of eight 2 KB slices, one for caching and the other seven for computing.

BLADE[43], CPU sends vector instructions to the cache, which are decoded by the cache manager and assigned to SRAM arrays for calculation. This centralized control flow becomes the bottleneck that limits the parallelism of in-cache computing, making it difficult to support multi-tasking parallel execution and overlap memory accesses with computations.

Secondly, we argue that it is unwise to perform the whole DNN inference in cache. The SRAM is efficient at in-situ vector operations but cannot conveniently support auxiliary functions such as reduction, non-linear activation, and quantization. Based on the design philosophy that different types of computation should be allocated to appropriate devices, we believe that more peripheral logic, and even a lightweight processor, should be added together to perform the auxiliary functions more efficiently, which is difficult to achieve in highly organized caches.

2.4 Motivation

From the above analysis, we find that the emerging SRAM computing technology can complement the limited performance of many-core nodes. The MIMD control mode of many-core also alleviates the control flow bottleneck that affects the parallelism of in-cache computing. Therefore, we propose a novel many-core architecture where each node couples a computing memory (CMem) and the RISC-V core together. This heterogeneous architecture brings three benefits.

First, from the perspective of CMem, the CMem is surrounded by a programmable core capable of handling computations which the CMem is not good at. This allows for the simplification of CMem design and leads to higher performance.

Second, from the viewpoint of the RISC-V core, CMem acts as a vector MAC unit managed by the scoreboard. The computing efficiency can be further improved by reasonable intra-node task allocation, which assigns vector MAC operations to CMem and leaves other auxiliary functions to the core.

Third, in terms of many-core, each node has its own control flow. The nodes can communicate with each other to achieve efficient data reuse and overlap of computation and memory access, providing higher computational efficiency compared to in-cache computing.

The subsequent parts of this paper will focus on these three points for optimization.

3 ARCHITECTURE DESIGN

3.1 Overview

The overall architecture of MAICC is shown in Figure 3(a). Each node in the many-core array is a lightweight RISC-V processor which supports RV32 IMA ISA, as shown in Figure 3(b). The pipeline has five stages and adopts in-order issue, out-of-order execution mechanism. The out-of-order execution is achieved by scoreboard algorithm to prevent multi-cycle instructions, such as idiv, remote request, and CMem extension instructions, from blocking the pipeline. The CMem is integrated into the pipeline to accelerate vector MAC operations.

The host multi-core CPU is a high-performance general-purpose processor. It runs the operating system and is responsible for resource management and task allocation of the many-core array. At the top and bottom of the array are two rows of last-level caches (LLC) connected to many-core DRAM. The many-core DRAM is striped, with each channel corresponding to one LLC node. The 2D mesh NoC uses X-Y routing to connect the host CPU, many-core arrays, and caches together.

To provide an agile method for cores to communicate with each other, we adopt remote load/store primitives. With this primitive, each core can directly access the many-core DRAM and other cores' data memory with a single load/store/AMO instruction. When a core issues a remote access instruction, it injects into the NoC a package containing 32-bit data. The receiver core processes this

Table 1: Global Virtual Address Mapping

Address Space	Range	Size	Description
Local address	0x00000000 - 0x000017FF	20KB (6KB visible)	0x00000000 - 0x00000FFF: 4KB local data memory 0x00001000 - 0x000017FF: 2KB CMem slice 0
Remote core address	0x40000000 - 0x7FFFFFFF	1GB(16KB each core)	32b address : 01xxxxxx_xxyyyyyy_yyoooooo_oooooooo x: x position, y: y position, o: offset
Many-core DRAM address	0x80000000 - 0xFFFFFFFF	2GB	The DRAM is uniformly divided into 32 channels connected with 32 last-level caches.

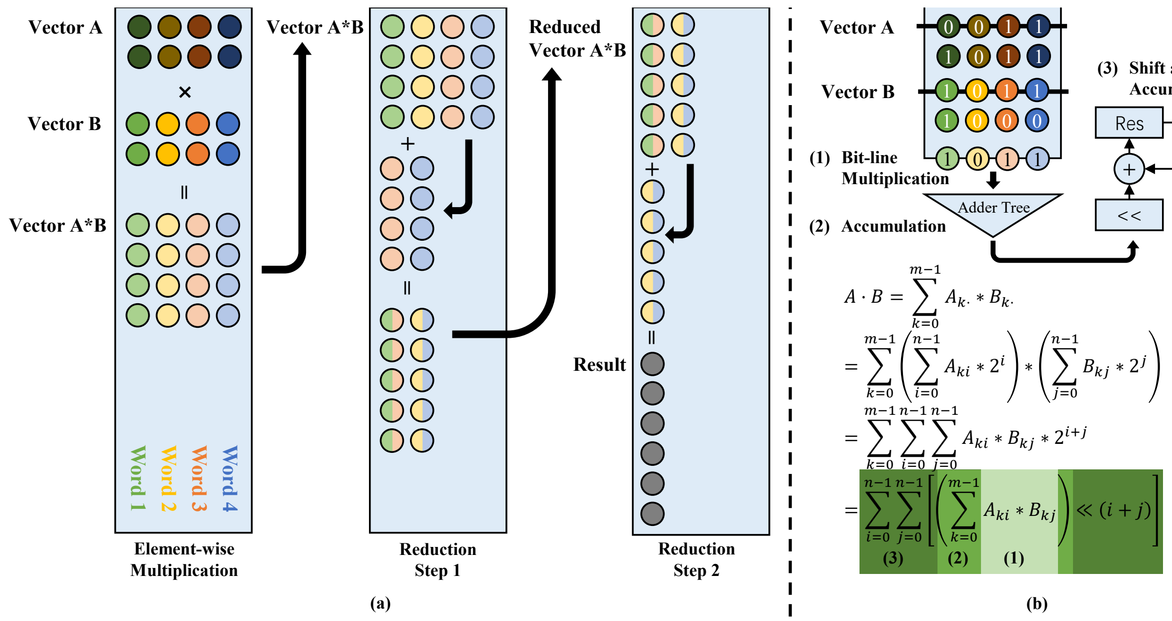
Vector A
Vector B
Vector A*B
Word 1
Word 2
Word 3
Word 4
Element-wise Multiplication
Vector A*B
Reduction Step 1
Reduced Vector A*B
Result
Reduction Step 2
(a)
Vector A
Vector B
(1) Bit-line Multiplication
(2) Accumulation
Adder Tree
(3) Shift and Accumulation
Res
<<
$A \cdot B = \sum_{k=0}^{m-1} A_{k\cdot} * B_{k\cdot}$
$= \sum_{k=0}^{m-1} \left(\sum_{i=0}^{n-1} A_{ki} * 2^i\right) * \left(\sum_{j=0}^{n-1} B_{kj} * 2^j\right)$
$= \sum_{k=0}^{m-1}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} A_{ki} * B_{kj} * 2^{i+j}$
$= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\left[\left(\sum_{k=0}^{m-1} A_{ki} * B_{kj}\right) \ll (i+j)\right]$
(3) (2) (1)
(b)

Figure 4: (a) Neural Cache [17] implements MAC temporally. (b) CMem implements MAC spatially.

package and replies to the sender if needed. The concurrency is ensured by address isolation at programming time.

To support remote access operations, we adopt partitioned global address space. For each core, the virtual address space is divided into three regions as shown in Table 1. Local address contains local data memory and CMem, remote core address corresponds to local memory of other nodes, and many-core DRAM address corresponds to the 2GB many-core DRAM.

3.2 Computing Memory

This section describes the structure of proposed computing memory and the calculation process of vector MAC operations. Compared with existing compute-in-cache works, we make two improvements to the CMem.

The first improvement is slicing. Existing works re-purpose the standard 8KB SRAM array as computing units, where each bit-line has 256 bits. However, bit-line computing uses only three of them, two for reading and the third for writing. For example, in Neural Cache, an SRAM array contains a filter of $3 * 3 * 256$ and an ifmap window of the same size, but 9 vector multiplications can only perform serially, resulting in low computational efficiency.

Instead, we partition the SRAM array horizontally into many slender slices, as shown in Figure 3(c). There are still 256 columns but fewer rows in one slice. Operations in different slices do not interfere with each other and thus can be parallelized. Given the same capacity, the more slices are partitioned, the more parallelism we get. However, more slices lead to shorter bit-lines and stricter data locality requirements. The overhead of data movement between slices is also increasing. We implement the CMem as eight slices of 2KB. Each slice includes 64 rows and 256 columns and can hold eight 8-bit or four 16-bit vectors. Slice 0 supports bi-directional reading and writing using 8-transistor storage cells [51] and is responsible for caching and transposing data. The remaining seven slices perform bit-serial computations in parallel.

The second improvement is hardware-implemented reduction. As mentioned in Section 2.1, matrix multiplication and convolution can be decomposed into vector MAC operations. Due to limited peripheral logic, existing bit-serial architectures have to write the calculation results back to the same bit-line. Therefore, they have to adopt element-wise primitives, in which calculation results of vectors are still vectors. Consequently, they must support a reduction operation to accumulate all items in one vector based only on element-wise primitives. As shown in Figure 4(a), Neural

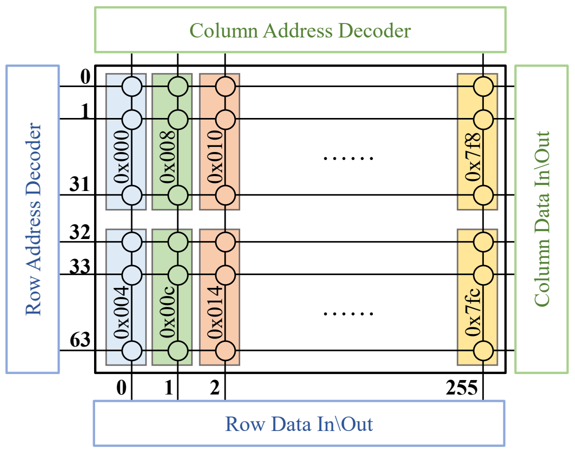

Figure 5: Address mapping of CMem slice 0.

Cache implements reduction as iterative shift and addition, which takes 8 (=$log_2 256$) iterations to accumulate all 256 items in one vector.

In this paper, we design a hardware-implemented vector MAC primitive. We transform the MAC operation as the equation in Figure 4(b). Assume A and B are m-element n-bit vectors stored in CMem. A_{ki} denotes the i-th bit of the k-th element in vector A, and B_{kj} denotes the j-th bit of the k-th element in vector B, where $0 \leq k < m$ and $0 \leq i, j < n$. The computation is composed of three steps. In step (1), the i-th row of vector A ($A_{\cdot i}$) and the j-th row of vector B ($B_{\cdot j}$) are activated simultaneously to get the AND values at the bit-lines. In step (2), the adder tree at the bottom accumulates the bits of all the 256 bit-lines. In step (3), the partial sum is shifted by $(i + j)$ and further accumulated into the *Res* register. These three steps work in a pipeline manner. After all the row pairs (i, j) are multiplied and accumulated, the result is obtained in the *Res* register. With this primitive, MAC operations can be finished in about n^2 cycles.

Compared with the element-wise primitives, our MAC primitives are more efficient in time and space. In terms of time, our primitive eliminates the reduction step, which takes up 23% of the computation cycles in Neural Cache. In terms of space, the results of element-wise primitives are also vectors that should be stored in SRAM. However, the results of our primitive are single numbers that can be directly put into registers of the lightweight core, thus leaving more SRAM space for storing weights and ifmaps. Although element-wise primitives have richer semantics, we argue that MAC primitive is sufficient to perform major computations in DNN inference, and the remaining operations can be executed in the lightweight core.

3.3 ISA Extensions and CMem Scheduling

In the previous section, we described the implementation details of CMem. In this section, we will show how to control its execution and transfer data through the proposed extended ISA.

The vectors should be stored and managed in a transposed way inside the CMem. Although the weights can be transposed in advance and loaded directly from DRAM, the feature maps still need to be transposed at runtime. Thus, slice 0 serves as the input buffer and transpose unit. As shown in Figure 5, the 2KB slice 0 is vertically addressed from 0 to 2048 bytes and accessed by basic load and store instructions. At the same time, it is also addressable by row indexing from 0 to 63 and accessible through the extended ISA. In this way, the transposed vectors can be obtained by first vertically writing into slice 0 through conventional *store* and then horizontally reading out by the extended *Move.C* instruction. The other seven slices are dedicated to computing and only support row indexing. Vectors in those slices are not byte-addressable and can only be accessed by the extended ISA.

Table 2: ISA extensions of computing memory.

Operation	Cycles	Meaning
MAC.C	n^2	MAC operations of two n-bit vectors in one slice.
Move.C	n	Move an n-bit vector between slices.
SetRow.C	1	Set one row to be all zero or one.
ShiftRow.C	2	Shift one row in the granularity of 32.
LoadRow.RC	1	Remote load one row from another node.
StoreRow.RC	1	Remote store one row to another node.

The extended instructions of CMem are shown in Table 2. *MAC.C* performs MAC of two n-bit vectors in the same slice and writes the result back to general registers of the RISC-V core, as described in Figure 4(b). It takes n^2 cycles to execute in CMem and other several cycles in the pipeline. *Move.C* moves vectors between slices, which takes n cycles. *LoadRow.RC* and *StoreRow.RC* move the transposed vector between multiple cores through NoC so that one vector only needs to be transposed once in its entire life cycle, instead of being transposed at each core. In order to ensure the atomicity and avoid long time occupation of CMem by communication, we design the atomic row-level operation in hardware while the vector-level atomicity is realized through software lock. *SetRow.C* sets all bits in one row as 0 or 1 and is used to clear data. *ShiftRow.C* shifts a row in 32-bit granularity and is used for vector alignment. It takes two cycles, one for reading and the other for writing. Finally, each slice is equipped with an 8-bit control status register (CSR) for masking, with each bit enabling 32 bit-lines. We choose 32 as the granularity because channel numbers in convolutional layers are mostly a multiple of 32.

The CMem serves as a multi-cycle computing component in the pipeline. The state of each slice is managed by the scoreboard. The extended ISA includes multi-cycle instructions with n^2 or n cycles, so it is necessary to schedule instruction sequences to avoid pipeline stalls and to improve the utilization of CMem.

The first approach is to dynamically schedule the instructions by hardware at runtime. We add a 2-item first-in-first-out (FIFO) instruction issue queue before CMem to avoid structural hazards. At the ID stage, the CMem instruction checks the issue queue and will be blocked if the queue is full. Otherwise, even if the CMem is busy, it can still move on and enter the issue queue without blocking the pipeline. Also, the register file has two write ports to avoid conflicts in the WB stage.

The second approach is to statically reorder the instructions at compile stage. After compilation, the latency and data dependency of each CMem instruction is determined. Therefore, we can

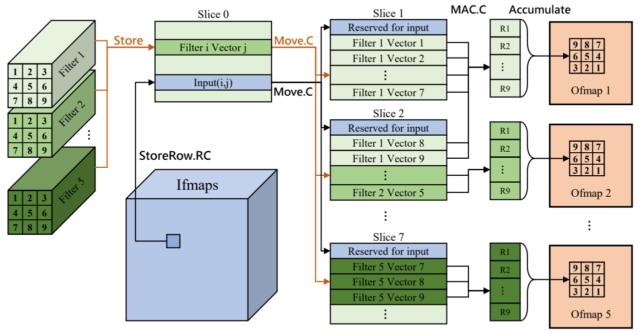

Figure 6: Data layout and computing flow of CMem. Each node keeps five 8-bit filters of size 3*3*256. R1-R9 are general-purpose registers of RISC-V and ofmaps are stored in data memory.

potentially fill the delay slots of CMem instructions by inserting data-independent instructions to reduce data hazards.

4 EXECUTION MODEL

In the previous section, we introduced the heterogeneous many-core architecture that integrates CMem to accelerate MAC computation. This section further discusses how to map DNN models into the proposed architecture to achieve high throughput and utilization rate. Firstly, We describe how convolutions, matrix multiplication, and auxiliary functions are allocated and executed inside one node. Then, we propose a novel inter-node streaming mechanism for intra-layer and inter-layer pipelining. And finally, we design a heuristic algorithm to segment and map the DNN model onto MAICC.

4.1 Intra-node Computing Flow

DNN models are typically composed of computational layers (CONV and FC) and auxiliary function layers (e.g., activation, pooling, normalization, and quantization). We fuse each computational layer with its subsequent auxiliary function layers into a mixed layer as the workload of each node. Specifically, CMem performs the computation layer, while the RISC-V pipeline executes auxiliary functions.

We take the CONV layer as an example to illustrate the data layout on CMem. The filters and ifmaps are split into $R*S$ and $H*W$ vectors along the channel dimension with each vector corresponding to all channels of one pixel. The vectors are transposed and placed in CMem for bit-serial MAC computation. Since CMem slices have 256 bit-lines, we assume the channel number $C = 256$ for convenience. For $C > 256$, the vectors can be divided into multiple sub-vectors of size 256, while for $C < 256$ (e.g., 32 or 64), we can place multiple vectors on the same word-lines and use $ShiftRow.C$ and CSR registers for shifting and masking data.

We adopt weight-stationary (WS) dataflow, which keeps the filters stationary in CMem while letting ifmap vectors enter CMem one by one to perform MACs. As shown in Figure 6, a filter of size $R*S*C$ with N-bit precision occupies $R*S*N$ rows in the CMem. These vectors can be scattered in different slices because

Algorithm 1: Pseudocode of computing core executing CONV layer.

```
for $t \leftarrow 1$ to $H*W$ do
    while $p \neq 1$ do
        do nothing;
    for $i \leftarrow 1$ to $7$ do
        $Move.C(Slice[0,0],\ Slice[i,0],\ N)$;
        for $j \leftarrow 1$ to $64/N$ do
            $psum[j] = MAC.C(Slice[i,0],\ Slice[i,Nj],\ N)$;
        accumulate $psum[]$ to the corresponding $ofmap[]$;
    while $nextp \neq 0$ do
        do nothing;
    for $i \leftarrow 0$ to $N-1$ do
        $Store.RC(Slice[0,i],\ NextSlice[0,i])$;
    $nextp = 1$;
    $p = 0$;
    if $ofmap[x,y,c]$ completes accumulation at last iteration then
        $res = pool(act(quant(norm(ofmap[x,y,c]))))$;
        $RemoteOfmap[x,y,c] = res$;
```

the accumulation of $R*S$ results of the same filter is performed in the pipeline rather than in-situ at the slice. Each slice reserves N rows for ifmap vectors, allowing the remaining $64-N$ rows for placing filters, so that one slice can hold $Q = \frac{64}{N} - 1$ vectors, and each node can place $\lfloor \frac{7*Q}{R*S} \rfloor$ filters at most.

Once the filters are in place, the ifmap vectors can be sequentially moved into the CMem. Each pixel of the ifmap, except for a few at the margins, is required to perform MACs with all $R*S$ filter pixels to get partial sums of an $R*S$ ofmap window. Therefore, in each iteration, only one ifmap pixel is moved into slice 0 and broadcasted to 7 slices for MACs. The partial sums obtained from CMem are accumulated into the ofmap stored in data memory by the core. In addition, the core is also responsible for executing auxiliary functions, accessing data, and communicating with other cores.

Following the above computing flow, a complete iteration takes $(7N + QN^2)$ cycles in CMem, while the core issues $7(Q+1)$ CMem instructions. Therefore, through the dynamic and static scheduling mentioned in Section 3.3, the core has the potential to utilize the free cycles to complete the accumulation and auxiliary functions, fetch the next ifmap vector, and store the ofmaps, thus enabling the overlap of CMem and the pipeline.

4.2 Inter-node Streaming

Previous works have proposed various dataflow solutions to accelerate DNN layers on accelerators or many-core architectures [9, 10, 21, 35]. They can be classified as weighted stationary (WS), output stationary (OS), and row stationary (RS), depending on the types of data that remain stationary. However, most traditional dataflows are implemented on PE arrays, in which data are moved and computed element by element due to limited storage and

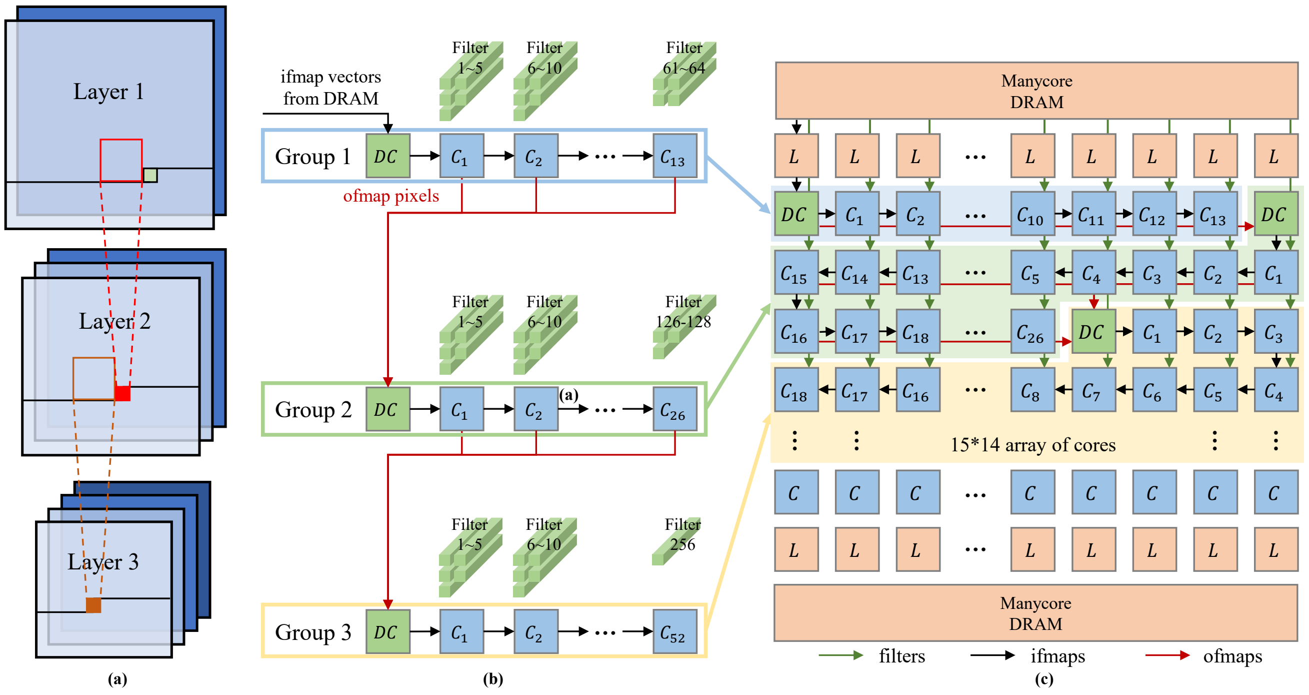

Figure 7: (a) Multi-layer parallel execution. (b) Ifmap vectors flow through node groups, where each computing core holds 5 filters. (c) The zig-zag mapping strategy. *DC*: Data Collection Core, C_i: computing core.

logic circuits of PE. When we introduce CMem which stores and computes in the granularity of vectors, some fine-grained dataflows such as OS and RS lack sufficient pipeline depth to gain efficiency, while WS still works.

We adopt WS to implement the multi-node streaming execution of CONV layers. As shown in Figure 7(b), a node group, corresponding to one CONV layer, consists of a data collection core and a sequence of computing cores. The data collection core is responsible for collecting ifmap vectors from DRAM or the previous node group, transposing and feeding them to the first computing core. Each computing core holds filters in its CMem, and the number of computing cores equals the number of filters divided by the number that a core can hold. Upon receiving one ifmap vector, the computing core performs the computing flow described in Section 4.1. In addition, the computing core passes the ifmap vector to the next core for ifmap reuse. When all ifmap vectors flow through the node group, all partial sums are accumulated into the entire ofmap. Different computing cores in the node group hold different filters and thus generate the corresponding channels of the ofmap.

The example code of computing core is shown in Algorithm 1. We use remote store primitives for inter-node communication, i.e., each core's ifmap vector is prepared by the previous core. H, W, and N denote the height, weight, and precision of ifmap. $Slice[i, j]$ and $NextSlice[i, j]$ denotes the i-th slice, j-th row of the local and the next computing core's CMem, and we store the ifmap vector in $Slice[0, 0 : (N-1)]$. $psum[]$ and $ofmap[]$ are stored in local data memory, while $RemoteOfmap[]$ resides in the remote memory of the next layer data collection core. p and $nextp$ are state variables that mark the validity of ifmap vectors for local and the next computing core. Line 2-3 checks if the local ifmap vector is ready, and p is released at line 14 after the ifmap vector is computed. Line 5 moves the ifmap vector from slice 0 to other slices, line 7 calls CMem to perform MACs, and line 8 accumulates partial sums to ofmaps. Line 9-10 checks $nextp$ and then line 11-13 sends the ifmap vector to the next core. Finally, line 15-17 executes the auxiliary functions and stores the ofmap to the data collection core of the next layer. Note that in each iteration we process the ofmap pixels completed in the previous iteration to avoid data dependency between CMem and the pipeline. In this way, we can reorder the instructions to overlap the computation of both.

Furthermore, the data locality of CONV layers can be exploited to achieve pipeline between adjacent layers. As shown in Figure 7(a), when the data collection core fetches and feeds ifmap vectors in a top-to-bottom, left-to-right order, the ofmap pixels are generated in the same order with a delay of R rows. In this way, once a new ofmap pixel is generated, it can be sent to the next node group immediately without waiting for other computations in this layer to complete. This allows the simultaneous mapping and execution of multiple layers in the many-core architecture through inter-layer pipelining, thus improving the utilization of many-core.

4.3 Layer Segmentation and Mapping

The node group is mapped to the many-core in a zig-zag way as shown in Figure 7(c). This mapping strategy ensures that two adjacent cores in the node group are also physically adjacent, leading to minimal ifmap transmission overhead. Also, the distance from computing cores to the next layer data collection core is relatively short, facilitating sending ofmaps.

Due to the limited on-chip storage, it is impossible to simultaneously map all layers of one DNN model to the many-core architecture, so it is necessary to divide the DNN model into segments. Simply putting as many layers as possible into one segment leads to workload imbalance between different core groups and reduces performance. Therefore, we need to model the latency of each layer. The expected running time of a computing core can be modeled as $H * W * T$, where the algorithm executes $H * W$ iterations and each iteration costs time T. In DNN models, the ifmap size $H * W$ is usually scaled down exponentially due to pooling layers, leading to a similar reduction of the expected running time. Based on this phenomenon, we design a heuristic algorithm that divides adjacent layers with the same ifmap size into one segment to achieve intra-segment workload balancing.

We further implement a mapping strategy for each segment to the many-core. Assuming Segment $S = \{1, 2, ..., L\}$ has L layers. We model the expected running time of each segment as the following optimization problem.

$$\begin{aligned} & min\ max_{i \in S}\ T_i \\ & s.t.\ T_i = max(T_i^{CMem}, T_i^{aux} + T_i^{rs}) \\ & T_i^{CMem} = k_1 n_i \\ & T_i^{aux} = k_2 n_i \\ & \sum_{i \in S} \frac{N_i}{n_i} < M \end{aligned} \tag{1}$$

where T_i^{CMem}, T_i^{aux}, and T_i^{rs} are the computing time of CMem, auxiliary functions, and remote storing ifmap vectors, respectively. N_i and n_i are the total number of filters and the number that one computing core holds. k_1 and k_2 are coefficients, and M is the many-core size. By optimizing this problem, we can get the optimal mapping scheme with the highest utilization.

5 EVALUATION METHODOLOGY

Baseline and Benchmark Setup. In the performance evaluation of MAICC, we use Intel i9-13900k as the CPU baseline and NVIDIA RTX 4090 as the GPU baseline. The specifications of the baselines are shown in Table 3. The benchmark program is the inference process of ResNet18 [25] with 8-bit quantization [27]. The batch size is set to 1. We do not include the first layer because it has very low parallelism with only 3 ifmap channels. We use PyTorch benchmark tools to evaluate the latency, while the power is estimated by RAPL [13] for CPU and nvidia-smi [37] for GPU. Since the inference latency of the quantized network is longer than the unquantized version, we use the unquantized version for comparison.

Table 3: Specifications of baseline CPU and GPU.

	CPU	GPU
Type	Intel Core i9-13900K	NVIDIA RTX 4090
Frequency	3.00GHz/2.20GHz	2.235 GHz
Cores	24	16384
Technology	10 nm	5 nm
Cache	Private 32KB L1I, 48 KB L1D, Private 2 MB L2, Shared 36 MB L3	128 KB per SM, 72 MB shared L2
Memory	64 GB DDR4	24 GB GDDR6X

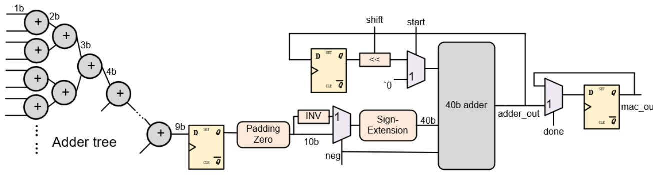

Figure 8: The peripheral circuits of CMem slice 1-7.

Table 4: Node Comparison.

	Scalar core	MAICC node	Neural Cache
Memory (KB)	20	20	40
Area (mm^2)	0.052	0.114	0.158
Energy (J)	1.03×10^{-4}	3.96×10^{-6}	4.03×10^{-6}
Cycles	1.24×10^7	59141	136416

Performance Model. To estimate the performance of MAICC, we develop a cycle-accurate simulator for the RISC-V pipeline and the CMem. It integrates booksim2 [29] and DRAMsim3 [34] to evaluate the latency of NoC and DRAM. Since CMem extended instruction set cannot be compiled automatically for now, we schedule the instructions manually in the computing node evaluation (Section 6.1), while assuming that CMem and the RISC-V pipeline can be fully overlapped in the overall evaluation (Section 6.3).

System Model. We implement the lightweight RISC-V pipeline in Verilog RTL. Assuming 28 nm technology and 1 GHz frequency, each core has 0.014 mm^2 area and 8 mW power. We also implement the SPICE model of the SRAM array with Virtuoso under the 40nm technology and 1.1V nominal voltage. Simulation is conducted with HSPICE to measure SRAM array power and latency. The peripheral circuits, as shown in Figure 8, are synthesized by Synopsys Design Compiler for area and power estimation. The energy consumptions for vertical write, Move.C, MAC.C, and remote load/store are estimated to be 4.75 pJ, 52.75 pJ, 28.25 pJ, and 53.01 pJ. The estimated area is 0.014 mm^2 for slice 0 and 0.023 mm^2 for slice 1-7. The energy and area are scaled down to 28nm, the same as the lightweight core.

The area and energy of data memory, last-level cache, and DRAM are evaluated by TSMC 65 nm Memory Compiler [46], McPAT [33], and DRAMsim3 [34], respectively. The NoC consumes 2.61 mm^2 area and 2.20 W static power, with a dynamic energy of 5.4 pJ per flit per hop, measured by dsent [45]. The total area of 210-core MAICC is estimated to be 28 mm^2.

6 RESULTS

6.1 Evaluation of Computing Node

We first evaluate the performance of one MAICC node. Table 4 shows the performance and overhead of the MAICC node, a baseline scalar core, and Neural Cache [17] under the same workload. The workload is a CONV layer that applies five filters of 3*3*256 to an

Table 5: Results of dynamic and static scheduling.

	One write-back port				Two write-back ports		
Queue Size	0	1	2	4	1	2	4
Cycles (w/o static)	61895	60761	59141	59141	60032	58250	58250
Cycles (with static)	52098	50802	50154	50154	50073	49263	49263

Table 6: Comparison of Layer Mapping Strategies.

		Single-layer		Greedy		Heuristic	
Index	Name	#Nodes	Latency(ms)	#Nodes	Latency(ms)	#Nodes	Latency(ms)
1	conv1_1	65	3.171	5		33	
2	conv1_2	65	3.171	5		33	
3	conv1_3	65	3.171	5		33	2.730
4	conv1_4	65	3.171	5		33	
5	shortcut	129	1.532	2		5	
6	conv2_1	129	1.803	8	9.022	16	
7	conv2_2	129	1.552	14		44	
8	conv2_3	129	1.552	14		44	
9	conv2_4	129	1.552	14		44	0.977
10	shortcut	129	0.404	4		8	
11	conv3_1	129	0.540	27		27	
12	conv3_2	129	0.437	53		53	
13	conv3_3	129	0.437	53		53	0.469
14	conv3_4	129	0.437	53	0.426	53	
15	shortcut	172	0.186	12		12	
16	conv4_1	172	0.300	172	0.300	172	0.300
17	conv4_2	208	0.203	208	0.203	208	0.203
18	conv4_3	208	0.203	208	0.203	208	0.203
19	conv4_4	208	0.203	208	0.203	208	0.203
20	linear	22	0.053	22	0.053	22	0.053
Total Latency (ms)			24.078		10.410		5.138

ifmap of 9*9*256. Compared with Neural Cache, MAICC node has higher computational and storage density with 43% latency and 50% memory size. The energy consumption of MAICC node is also slightly lower than Neural Cache.

Table 5 shows the impact of dynamic and static scheduling on the latency of MAICC node. The workload here precludes communication and focuses only on the computational process. Dynamic scheduling involves adding an instruction issue queue and an additional write-back port, while static scheduling refers to reordering the CMem instructions to hide latency during compilation. We discover that a relatively small queue size of 2 brings evident performance improvement of 4%. The 2-entry issue queue can eliminate the structural hazard brought by CMem instructions at the ID stage and adding more entries brings no more latency benefits. Adding a write-back port is also able to reduce the running time by 2%, indicating that the width of instruction commits is also the bottleneck. Static scheduling has 16% optimization because it hides the latency of multi-cycle CMem instructions by inserting independent instructions to avoid data-dependent instructions from blocking the pipeline to the maximum extent.

6.2 Layer Mapping Strategy Analysis

This section evaluates the proposed layer segmentation strategy. Table 6 shows the inference latency of ResNet18 under three segmentation strategies. The single-layer strategy does not segment the network and maps each layer separately into the many-core. The greedy strategy puts as many layers as possible into one segment, while the heuristic strategy refers to the algorithm presented in section 4.3. The segmentation results are marked by the horizontal lines in the table. The greedy strategy divides layers 1-12 and 13-15 into two segments, while the heuristic strategy divides layers 1-6, 7-11, and 12-15 into three segments. For each layer in layers 17-19, all the strategies divide it into a separate segment due to their large filter size which occupies at least 208 nodes. Results show that the single-layer strategy performs the worst, requiring 24 ms for inference, while the heuristic strategy only takes 5 ms, equivalent to 200 samples per second.

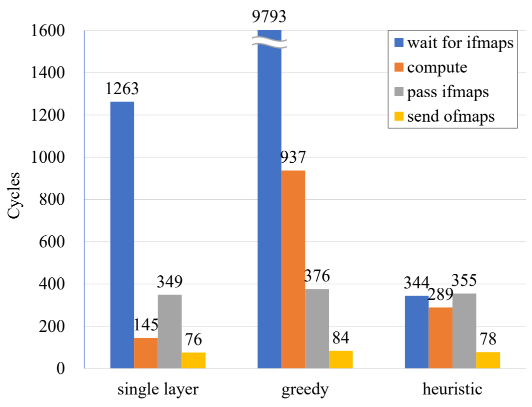

Figure 9: Time breakdown per iteration of layer 9.

Table 7: Overall Performance of MAICC and baselines on ResNet18.

	CPU	GPU	MAICC
Latency (ms)	22.3	1.02	5.13
Throughput (samples/s)	44.8	980.3	194.9
Average Power (W)	176.4	228.6	24.67
Throughput per Watt	0.25	4.29	7.90

Segmentation and mapping strategies have substantial effects on latency. Assigning too many (single-layer strategy) or too few (greedy strategy) computing cores to one layer both lead to performance degradation. We select one intermediate computing core of layer 9 and present its cycle breakdown per iteration, as shown in Figure 9. We discover that cycles to send ifmap vectors and ofmap pixels are stable under different mapping strategies, and cycles to perform computations are approximately inversely proportional to the number of nodes allocated to the layer, as expected. However, in the single-layer and greedy strategies, waiting for ifmap vectors takes up a large portion of cycles. In the single-layer strategy, a maximum number of nodes is assigned to one layer. Therefore, each core performs very few computations, while the communication overhead gets large, resulting in a bottleneck of communication and loading ifmap vectors. In the greedy strategy, although the computation amount of each core is sufficient, the computation speed of different layers varies greatly. Layers with high computation speed have to spend a long time waiting for data to flow out from the slower layers, which substantially reduces the utilization.

The comparison of greedy strategy segment 2 (layers 13-15) and heuristic strategy segment 3 (layers 12-15) reveals that adding

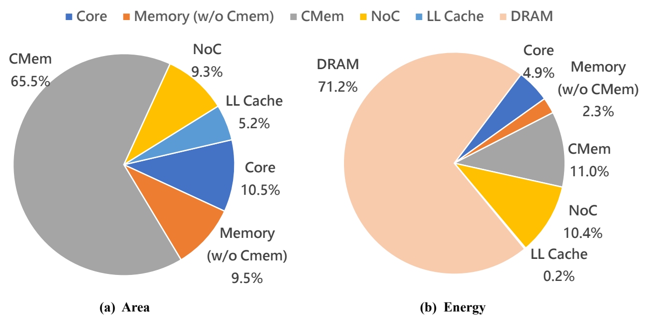

Figure 10: Area and Energy breakdown of MAICC.

one layer in the segment brings only 10% additional latency. This is obtained by the highly pipelined inter-layer mechanism. For example, in segment 3 of the heuristic algorithm, 83% of the computation time of layer 12 overlaps with layer 15. Furthermore, The filter load phase before computation can also start in batches to fully utilize the data locality of LLC, and in most cases the filter load phase takes no more than 10% of the total time.

6.3 Overall Performance of MAICC

This section evaluates the overall performance of MAICC on ResNet18 [25]. As shown in Table 7, the 210-core MAICC has a 4.3× speedup over CPU. Although MAICC has 0.2× the throughput of the GPU, this is because MAICC has only 210 scalar cores and less than 6 MB on-chip memory, while the GPU has 16,384 cuda cores and 88 MB on-chip cache (16 MB L1 cache and 72 MB L2 cache). Ideally assuming that MAICC's performance scales linearly with the number of cores, MAICC can potentially achieve 2.9× better performance compared to the GPU with the same on-chip memory capacity. In addition, we choose a conservative frequency of 1GHz as bit-line computing requires longer latency than conventional memory accesses[17].

Table 7 also presents the power consumption and computing efficiency measured by throughput/Watt. MAICC is able to improve the computing efficiency by 31.6× on CPU and 1.8× on GPU. This improvement mainly comes from the efficient data reuse brought by dataflow. DNN models are mapped onto MAICC by segment. Only the inputs and outputs of the segment need to access DRAM, while the fmaps of intermediate layers move only between on-chip scratchpads.

We also compare the computational efficiency of MAICC with Neural Cache [17], a prior in-SRAM design. Using Inception-v3 [48] as workloads, Neural Cache reached 22.90 GFLOPS/W without modeling DRAM, while MAICC has a computational efficiency of 50.03 GFLOPS/W (2.2x better) on ResNet18 after excluding the power consumption of DRAM.

Figure 10 shows the area and energy breakdown of the MAICC. 65% of the on-chip area is occupied by CMem, of which about one-third is the computing logic (the adder tree) and the remaining two-thirds are memory cells. The core and the on-chip memory (not including CMem) consume 11% and 10%, followed by the NoC (9%) and the LL Cache (5%). The energy consumption of MAICC comes primarily from DRAM access with 71%, followed by CMem and NoC, each occupying 11%. The area of the RISC-V core and on-chip memories occupies less than 10%.

7 RELATED WORK

NN Accelerators. In recent years, NN accelerators have been extensively explored with the dominance of neural networks. Early NN accelerator[9, 20, 30, 35] used simple PEs to implement basic fixed-point MAC operations and supported limited sets of simple auxiliary functions such as ReLU and max pooling. With the pursuit of generalization, recent NN accelerators [18, 50] have adopted configurable PEs that support multiple fixed-point and floating-point precision, and special functional units (SFU) that feature dozens of auxiliary functions.

Meanwhile, to enable multi-task acceleration and address the under-utilization caused by the mismatch between PE array sizes and network shapes, existing works [3, 11, 20, 21, 32, 53] propose tiled PE architectures and explore efficient task partitioning and scheduling strategies to decompose DNN workloads and match the granularity of tiles.

On the other hand, dataflows based on 2D PE arrays have also been deeply developed. Various kinds of efficient dataflows, such as WS, OS, and RS, have been proposed. Tangram [21] further discusses the combination of two dataflows and the mapping strategies of multiple NN layers onto PE array tiles.

Despite these developments, NN accelerators are still unable to achieve the same generality and compilability as general-purpose processors and thus can only be used as co-processors to accelerate computations. In contrast, many-core architectures can leverage the dataflow technology to obtain high energy efficiency while keeping generality.

Many-core execution framework. The execution and scheduling mechanism of many-core architectures is a problem that still requires comprehensive research. [10] extends more than 100 dense or sparse operators of the PyTorch framework on a 128-core architecture. It enables the decoupling of computation and access through prefetching of custom access accelerators and is able to run the inference and training process of neural networks efficiently. However, it only analyzes and optimizes the execution process of a single operator at one time and does not support the case of multiple operators mapped to many-core.

Esperanto[15] integrates vector processing units in many-core architecture and achieves extreme energy efficiency with low-voltage technology. It supports common machine learning frameworks and generates executable code through GLOW[42] and TVM [8].

Rockcress [4] re-configures multiple scalar cores in a many-core architecture into vector execution lanes and designs the instruction forwarding network to deliver vector instructions. When Rockcress runs in vector mode, the scalar cores skip the instruction fetch phase and fetch instructions from the NoC. It also uses the decoupled access/execution mechanism to hide access latency and improve efficiency. On the other hand, this paper implements vector MAC instructions within the scalar core by introducing computing memory, and the executing order between different cores is more loosely restricted.

8 CONCLUSION

The growing complexity and diversity of DNN models have spurred the need for a generic many-core architecture. Faced with the dilemma of tight energy and area constraints and high-performance demand of many-core nodes, the emerging cache-based in-memory-computing technology re-purposes the cache of lightweight small cores for low-precision vector MAC computation. In this paper, we propose MAICC, a many-core architecture integrated with in-cache computing, which utilizes the computing memory to implement the computational-intensive CONV and FC operators in DNNs, and scalar RISC-V cores to implement the subsequent accumulation and auxiliary functions for efficient intra-node parallelism. We also propose a complete execution mechanism for MAICC, which divides DNNs into segments and maps multiple layers in a segment onto MAICC simultaneously, thus enabling intra-layer and inter-layer pipelining. Evaluation results show that the 210-core MAICC achieves a 4.3x performance and 31.6x energy efficiency over CPUs. MAICC also achieves a 1.8x energy efficiency with only 4MB of on-chip storage compared to the GPU. The MIMD execution mode of MAICC has the potential to support parallel inference of multiple DNN models and our future work will focus on the scheduling, mapping, and optimization of multi-model parallel inference.

ACKNOWLEDGMENTS

This work was supported in part by the National Key Research and Development Program of China under grant No.2020YFB1600200, in part by National Natural Science Foundation of China (NSFC) under Grant No.62250006, 62072266, and 62204271, in part by Tsinghua University Initiative Scientific Research Program, in part by Suzhou-Tsinghua Innovation Leadership Program, in part by Tsinghua-Toyota Joint Research Fund, and in part by Beijing National Research Center for Information Science and Technology (BNR2022RC01003).

REFERENCES

[1] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute Caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 481–492.

[2] Mrinal R Bachute and Javed M Subhedar. 2021. Autonomous Driving Architectures: Insights of Machine Learning and Deep Learning Algorithms. *Machine Learning with Applications* 6 (2021), 100164.

[3] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. 2020. A Multi-Neural Network Acceleration Architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 940–953.

[4] Philip Bedoukian, Neil Adit, Edwin Peguero, and Adrian Sampson. 2021. Software-Defined Vector Processing on Manycore Fabrics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 392–406.

[5] Brent Bohnenstiehl, Aaron Stillmaker, Jon J Pimentel, Timothy Andreas, Bin Liu, Anh T Tran, Emmanuel Adeagbo, and Bevan M Baas. 2017. KiloCore: A 32-nm 1000-Processor Computational Array. *IEEE Journal of Solid-State Circuits* 52, 4 (2017), 891–902.

[6] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. 2019. Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 2 (2019), 530–543.

[7] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 269–284.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[9] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 367–379.

[10] Lin Cheng, Peitian Pan, Zhongyuan Zhao, Krithik Ranjan, Jack Weber, Bandhav Veluri, Seyed Borna Ehsani, Max Ruttenberg, Dai Cheol Jung, Preslav Ivanov, et al. 2021. A Tensor Processing Framework for CPU-Manycore Heterogeneous Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 6 (2021), 1620–1635.

[11] Yujeong Choi and Minsoo Rhu. 2020. PREMA: A Predictive Multi-task Scheduling Algorithm for Preemptible Neural Processing Units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 220–233.

[12] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv preprint arXiv:1412.3555* (2014).

[13] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory Power Estimation and Capping. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*. 189–194.

[14] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, et al. 2018. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro* 38, 2 (2018), 30–41.

[15] Dave Ditzel, Roger Espasa, Nivard Aymerich, Allen Baum, Tom Berg, Jim Burr, Eric Hao, Jayesh Iyer, Miquel Izquierdo, Shankar Jayaratnam, et al. 2021. Accelerating ML Recommendation with over a Thousand RISC-V/Tensor Processors on Esperanto's ET-SoC-1 Chip. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 1–23.

[16] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *arXiv preprint arXiv:2010.11929* (2020).

[17] 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 *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 383–396.

[18] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–14.

[19] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2019. Duality Cache for Data Parallel Acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture*. 397–410.

[20] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. Tetris: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 751–764.

[21] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. Tangram: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 807–820.

[22] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W Mahoney, and Kurt Keutzer. 2021. AI and Memory Wall. *RiseLab Medium Post* (2021).

[23] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 580–587.

[24] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics* 37, 3 (2020), 362–386.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[26] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[27] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2704–2713.

[28] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. 2016. A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory. *IEEE Journal of Solid-State Circuits* 51, 4 (2016), 1009–1021.

[29] Nan Jiang, Daniel U Becker, George Michelogiannakis, James Balfour, Brian Towles, David E Shaw, John Kim, and William J Dally. 2013. A Detailed and

Flexible Cycle-Accurate Network-on-Chip Simulator. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 86–96.

[30] 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.

[31] Mingu Kang, Eric P Kim, Min-sun Keel, and Naresh R Shanbhag. 2015. Energy-efficient and High Throughput Sparse Distributed Memory Architecture. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2505–2508.

[32] HT Kung, Bradley McDanel, Sai Qian Zhang, Xin Dong, and Chih Chiang Chen. 2019. Maestro: A Memory-on-Logic Architecture for Coordinated Parallel Use of Many Systolic Arrays. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Vol. 2160. IEEE, 42–50.

[33] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 469–480.

[34] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: A cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 106–109.

[35] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 553–564.

[36] Enrique Marti, Miguel Angel De Miguel, Fernando Garcia, and Joshue Perez. 2019. A Review of Sensor Technologies for Perception in Automated Driving. *IEEE Intelligent Transportation Systems Magazine* 11, 4 (2019), 94–108.

[37] Nvidia Corporation. [n. d.]. *"Nvidia System Management Interface."*. https://developer.nvidia.com/nvidia-system-management-interface, accessed 20 April 2023..

[38] Andreas Olofsson. 2016. Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip. *arXiv preprint arXiv:1610.01832* (2016).

[39] Kariofyllis Patsidis, Chrysostomos Nicopoulos, Georgios Ch Sirakoulis, and Giorgos Dimitrakopoulos. 2020. RISC-V^2: A Scalable RISC-V Vector Processor. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.

[40] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 779–788.

[41] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *Advances in Neural Information Processing Systems* 28 (2015).

[42] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *arXiv preprint arXiv:1805.00907* (2018).

[43] William Andrew Simon, Yasir Mahmood Qureshi, Marco Rios, Alexandre Levisse, Marina Zapater, and David Atienza. 2020. BLADE: An in-Cache Computing Architecture for Edge Devices. *IEEE Trans. Comput.* 69, 9 (2020), 1349–1363.

[44] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[45] Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. 2012. DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling. In *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*. IEEE, 201–210.

[46] Synopsys, Inc. [n. d.]. *"Synopsys Memory Compilers."*. https://www.synopsys.com/dw/ipdir.php?ds=dwc_sram_memory _compilers, accessed 20 April 2023..

[47] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.

[48] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2818–2826.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *Advances in Neural Information Processing Systems* 30 (2017).

[50] Swagath Venkataramani, Vijayalakshmi Srinivasan, Wei Wang, Sanchari Sen, Jintao Zhang, Ankur Agrawal, Monodeep Kar, Shubham Jain, Alberto Mannari, Hoang Tran, et al. 2021. RaPiD: AI Accelerator for Ultra-low Precision Training and Inference. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 153–166.

[51] Jingcheng Wang, Xiaowei Wang, Charles Eckert, Arun Subramaniyan, Reetuparna Das, David Blaauw, and Dennis Sylvester. 2019. A 28-nm Compute SRAM With Bit-Serial Logic/Arithmetic Operations for Programmable In-Memory Vector Computing. *IEEE Journal of Solid-State Circuits* 55, 1 (2019), 76–86.

[52] Zhangjing Wang, Yu Wu, and Qingqing Niu. 2019. Multi-Sensor Fusion in Automated Driving: A Survey. *IEEE Access* 8 (2019), 2847–2868.

[53] Ahmet Caner Yüzügüler, Canberk Sönmez, Mario Drumond, Yunho Oh, Babak Falsafi, and Pascal Frossard. 2023. Scale-out Systolic Arrays. *ACM Transactions on Architecture and Code Optimization* 20, 2 (2023), 1–25.

[54] Florian Zaruba, Fabian Schuiki, and Luca Benini. 2020. Manticore: A 4096-Core RISC-V Chiplet Architecture for Ultraefficient Floating-Point Computing. *IEEE Micro* 41, 2 (2020), 36–42.

Received 28 April 2023; revised 7 July 2023; accepted 18 September 2023