VisionAGILE: A Versatile Domain-Specific Accelerator for Computer Vision Tasks

Bingyi Zhang, Rajgopal Kannan, Carl Busart, and Viktor K. Prasanna, *Life Fellow, IEEE*

***Abstract*—The emergence of diverse machine learning (ML) models has led to groundbreaking revolutions in computer vision (CV). These ML models include convolutional neural networks (CNNs), graph neural networks (GNNs), and vision transformers (ViTs). However, existing hardware accelerators designed for CV lack the versatility to support various ML models, potentially limiting their applicability to real-world scenarios. To address this limitation, we introduce VisionAGILE, a domain-specific accelerator designed to be versatile and capable of accommodating a range of ML models, including CNNs, GNNs, and ViTs. VisionAGILE comprises a compiler, a runtime system, and a hardware accelerator. For the hardware accelerator, we develop a novel unified architecture with a flexible data path and memory organization to support the computation primitives in various ML models. Regarding the compiler design, we develop a unified compilation workflow that maps various ML models to the proposed hardware accelerator. The runtime system executes dynamic sparsity exploitation to reduce inference latency and dynamic task scheduling for workload balance. The compiler, the runtime system, and the hardware accelerator work synergistically to support a variety of ML models in CV, enabling low-latency inference. We deploy the hardware accelerator on a state-of-the-art data center FPGA (Xilinx Alveo U250). We evaluate VisionAGILE on diverse ML models for CV, including CNNs, GNNs, hybrid models (comprising both CNN and GNN), and ViTs. The experimental results indicate that, compared with state-of-the-art CPU (GPU) implementations, VisionAGILE achieves a speedup of $81.7\times$ ($4.8\times$) in terms of latency. Evaluated on standalone CNNs, GNNs, and ViTs, VisionAGILE demonstrates comparable or higher performance with state-of-the-art CNN accelerators, GNN accelerators, and ViT accelerators, respectively.**

***Index Terms*—Compiler, computer architecture, computer vision, domain-specific accelerator, runtime system.**

I. Introduction

THE emergence of diverse machine learning (ML) models has brought about a significant revolution in computer vision (CV), facilitating various novel CV applications. These ML models include convolutional neural networks (CNNs) [1], [2], [3], graph neural networks (GNNs) [4], [5], [6], and vision transformers (ViTs) [7], [8]. While various ML models are available for CV tasks, there is no one-size-fits-all solution, as different models have different strengths. *Convolutional Neural Networks (CNNs):* CNNs excel in tasks like image classification [1], object detection [3], and image segmentation [9]. They are good at capturing local patterns and hierarchical features through their convolutional layers. CNNs are well-suited for handling large image datasets, offering computational efficiency. However, they are less effective when dealing with graph-structured data or sequences and struggle to model long-range dependencies. *Graph Neural Networks (GNNs):* GNNs are designed to capture relationships and propagate information in graph data. They are an ideal choice for CV tasks where the data structure is defined by nodes and edges, as seen in point clouds [10] and 3-D meshes. However, GNNs are less suitable for directly handling regular grid-like data, such as images. *Vision Transformers (ViTs):* ViTs are tailored for image analysis tasks similar to CNNs. However, they adopt a different approach by leveraging self-attention mechanisms to model image content. Unlike CNNs, ViTs are scalable and good at capturing long-range dependencies in images. Nevertheless, ViTs require a substantial amount of labeled data for training, which can be resource-intensive. They can be computationally expensive when compared to specific CNN architectures. Additionally, ViTs depend on positional encoding in regular data and cannot directly deal with graph-structured data.

Different ML models have different strengths, necessitating the selection between CNNs, GNNs, or ViTs for real-world applications based on specific problem requirements and data structures. Furthermore, many CV tasks leverage the combined strengths of different models. For example, GNN-based CV tasks utilize the capabilities of both CNNs and GNNs to enable a wide array of innovative CV applications [11], [12]. Several studies [13], [14] integrate CNNs and ViTs to enhance model robustness in CV tasks. Extrapolating the current research trend, we anticipate that computer vision systems, such as autonomous driving cars, will leverage a diverse range of ML models and various combinations of models. Moreover, in cloud computing platforms provided by service providers [15], [16], [17], the capability to support a diverse range of machine learning models is crucial to meet the demands of their customers. Consequently, developing a versatile domain-specific accelerator (DSA) that supports diverse ML models, including CNNs, GNNs, and ViTs, becomes essential.

Received 23 April 2024; revised 8 September 2024; accepted 12 September 2024. Date of publication 24 September 2024; date of current version 17 October 2024. This work was supported in part by DEVCOM Army Research Lab (ARL) under Grant W911NF2220159 and in part by the National Science Foundation (NSF) under Grant CCF-1919289 and Grant OAC-2209563. Equipment and support by AMD AECG are greatly appreciated. Recommended for acceptance by J. Becker. *(Corresponding author: Bingyi Zhang.)*

Bingyi Zhang and Viktor K. Prasanna are with the Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, CA 90089 USA (e-mail: bingyizh@usc.edu; prasanna@usc.edu).

Rajgopal Kannan and Carl Busart are with DEVCOM Army Research Office, Los Angeles, CA 90089 USA (e-mail: rajgopal.kannan.civ@army.mil; carl.e.busart.civ@army.mil).

Digital Object Identifier 10.1109/TPDS.2024.3466891

Several domain-specific accelerators [18], [19], [20], [21], [22], [23] have been developed to enhance the performance of computer vision (CV) tasks, such as Xilinx DPU [18], DLA [19], OPU [20], Google TPU [21] and TPUv4 [24], and GraphAGILE [22]. However, prior DSAs are primarily designed to accelerate only one or two specific types of ML models, lacking the versatility to accommodate a broad range of ML models in CV, including CNNs, GNNs, and ViTs. General-purpose processors like CPUs and GPGPUs can support various ML models, but they often deliver suboptimal performance for these models due to several reasons: (1) General-purpose processors have complex cache hierarchies, leading to large data access latency for ML models. (2) The data path of CPUs is designed to handle complex control flow, making it inefficient for computation-intensive workloads. (3) The design of computation parallelism in general-purpose processors is not well-suited for handling irregular computation patterns in the models for graph structured data, such as GNNs. In summary, while general-purpose processors offer flexibility, they meet performance challenges when dealing with various ML models.

This paper proposes VisionAGILE, a novel versatile domain-specific accelerator to support a variety of ML models for computer vision (CV) tasks, including CNNs, GNNs, and ViTs. Our main contributions are summarized as follows:

- We propose VisionAGILE, a domain-specific accelerator on FPGA, to support a variety of ML models for CV tasks, including CNNs, GNNs, and ViTs.
- We develop a novel unified hardware architecture with a flexible data path and memory organization to support various computation primitives in CNNs, GNNs, and ViTs. The proposed hardware architecture is designed with a customized instruction set to enable software-like programmability.
- We develop a novel compiler design for the proposed hardware architecture, including:
  – a generic intermediate representation (IR) to represent various ML models.
  – a unified compilation workflow to map various ML models to the proposed hardware architecture.
  – the bytecode mechanism that enables the dynamic sparsity exploitation at runtime.
- We develop a novel runtime system that comprehensively leverages the proposed architecture design which supports both sparse and dense computation primitives. The proposed runtime system utilizes a comprehensive dynamic sparsity exploitation strategy that is capable of exploiting the data sparsity in the inference process to reduce the inference latency.
- We deploy the proposed accelerator on a state-of-the-art FPGA board – Xilinx Alveo U250. We evaluate VisionAGILE on diverse CV tasks that involve CNNs, GNNs, and ViTs. Compared with state-of-the-art CPU (GPU) implementations, VisionAGILE achieves a speedup of $81.7\times$ ($4.8\times$), respectively. When evaluated on standalone CNNs, GNNs, and ViTs, VisionAGILE demonstrates comparable performance with state-of-the-art CNN accelerators, GNN accelerators, and ViT accelerators.

The rest of this paper is structured as follows: Section II introduces the related work; Section III provides an overview of VisionAGILE; Section VI elaborates on the hardware design; Section V delves into the details of the compiler design; Section VII describes the proposed runtime system; Section VIII introduces the implementation details; Section VIII demonstrates the evaluation results;

II. Related Work

Many hardware accelerators have been proposed to accelerate convolutional neural networks (CNNs), graph neural networks (GNNs), or vision transformers (ViTs). However, no generic domain-specific accelerator supports all of these models. We introduce the related work as follows:

Domain-specific accelerators (DSAs) for CNNs: While many DSAs [18], [19], [20], [21], [25], [26], [27] exist for CNNs, such as AMD DPU [18], Google TPU [21], Dadiannao [26], and OPU [20], none are designed to support various ML models, including GNNs, and ViTs. The compiler design and the hardware architecture of CNN-based DSAs only support CNNs. For example, in CNNs, the primary computation kernel is 2D convolution, and Google TPU [21] utilizes the 2-D systolic array to execute the 2-D convolution operation. However, the 2-D systolic array cannot efficiently support graph message passing in GNNs.

Hardware accelerators for GNNs: Many hardware accelerators [22], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37] are proposed for GNNs. Some studies design hardware accelerators [28], [30], [31], [35] for specific GNN models, such as graph convolutional network [4]. However, these works focus on designing hardware architecture optimized for sparse matrix multiplication in GNNs, and their architectures are not efficient for convolution operations in CNNs or multi-head self-attention in ViTs. Several works developed automatic frameworks to generate accelerators per input graph [32] or per GNN model [37]. However, these studies require regenerating the optimized accelerator if the input graph or GNN model changes. Moreover, their frameworks do not support CNNs or ViTs.

Hardware accelerators for ViTs: Many hardware accelerators [38], [39], [40], [41], [42] are proposed for ViTs. However, prior works either focus on designing algorithm-hardware codesigns [38], [40] for ViTs or developing design automation frameworks or generated quantized ViTs models and the corresponding hardware accelerators [42]. Their algorithmic optimizations and hardware design are exclusively for ViTs and lack the support for CNNs or GNNs.

Prior researches have focused primarily on developing DSAs or hardware accelerators for specific models. However, a notable gap remains in the availability of a versatile DSA capable of supporting various models, including CNNs, GNNs, and ViTs. In many real-world scenarios, there is a growing demand for versatile DSAs that can effectively accommodate a variety of models. Take, for instance, the field of autonomous driving, where a wide range of input data modalities, such as images [43], videos [44], and point clouds [10], require the deployment of various models to ensure efficient data processing. Given the

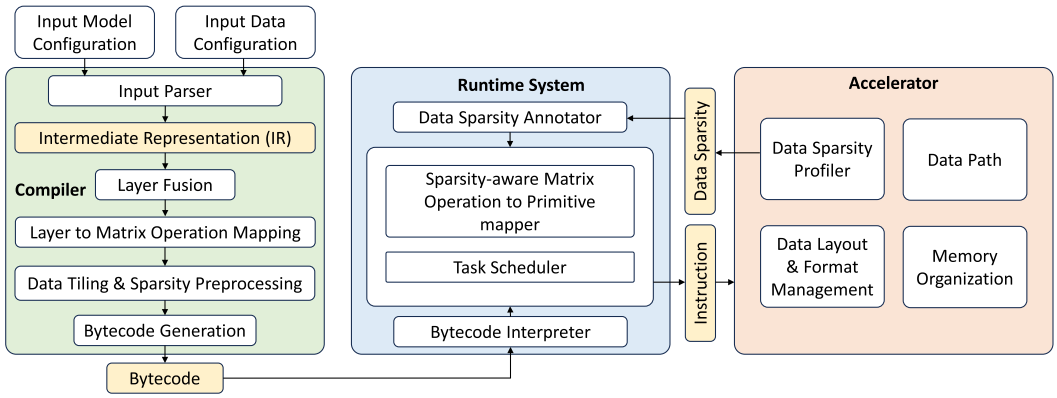


Fig. 1. Overview of VisionAGILE. VisionAGILE consists of the compiler (Section V), runtime system (Section VII), and the accelerator (Section VII). The overview of the accelerator is demonstrated in Fig. 2.

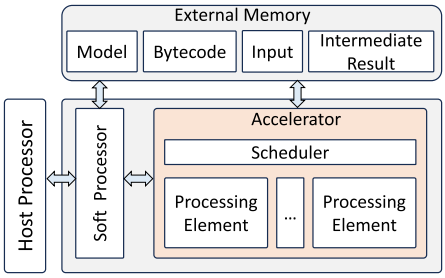


Fig. 2. Overview of accelerator.

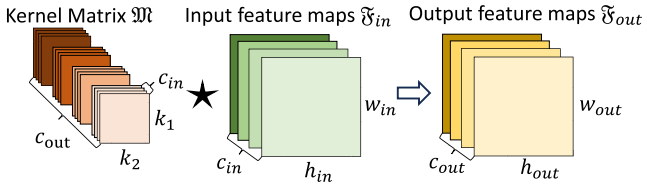


Fig. 3. Diagram of convolutional (Conv) layer.

ever-expanding applications of diverse machine learning models across various computer vision tasks, it becomes clear that there is an urgent need for a DSA capable of accommodating this rich diversity.

III. Overview

Fig. 1 depicts the key components of VisionAGILE, which consists of a compiler, a runtime system, and a hardware accelerator. Fig. 2 depicts the hardware platform to execute VisionAGILE, which consists of a host processor to execute the compiler, a soft processor to execute the runtime system, and a hardware accelerator. The runtime system and the accelerator are tightly coupled through a low-latency interconnection for efficient data communication.

1) Compiler: The compiler is executed on the host processor. It takes as input both the model configuration and the data configuration, translating them into *intermediate representation (IR)* (Section V-A). This intermediate representation functions as a high-level abstraction for the input model. Subsequently, the compiler performs a series of steps to produce bytecode for the runtime system. This bytecode is not directly executable by the hardware accelerator; instead, it acts as a bridge between the high-level intermediate representation and the low-level hardware instructions (computation primitives). The rationale behind employing bytecode is rooted in the runtime system's ability to exploit dynamic sparsity for mapping matrix operations to low-level computation primitives based on data sparsity. Since the level of data sparsity in intermediate results remains unknown during the compilation phase, the compiler generates bytecodes that does not depend on data sparsity information.

2) Runtime System: The runtime system is executed on the soft processor. It takes the sparsity-independent bytecode (See Section V-E) as input. The data sparsity annotator obtains the data sparsity information of intermediate results from the accelerator (through a Sparsity Profiler, See Section VI-C1) and annotates the data sparsity information in the bytecode. Then, the runtime system performs just-in-time (JIT) compilation to translate the bytecode into low-level instructions for hardware execution. During runtime, the runtime system performs two optimizations to reduce the inference latency: (1) sparsity-aware matrix operation to primitive mapping (Section VII-A) to exploit the data sparsity for reducing computation complexity and (2) dynamic task scheduling for workload balance (Section VII-B).

3) Accelerator: The hardware accelerator is equipped with a flexible data path and memory organization to support various computation primitives in CNNs, GNNs, and ViTs. Given that different computation primitives require varying data layouts and formats, the accelerator incorporates efficient hardware mechanisms for the transformation of both the data layout and the format. Furthermore, in light of the runtime system's need for intermediate result data sparsity, the accelerator is equipped with a data sparsity profiler (Section VI-C1) to quickly profile the data sparsity information of these intermediate results.

IV. Compiler Design

This Section introduces the compiler design in detail, including (1) the intermediate representation to represent the computation graph of the input model, (2) the unified compilation workflow to map various models to the hardware accelerator,

TABLE I
INTERMEDIATE REPRESENTATION (IR) OF THE COMPUTATION LAYERS IN CNNS, GNNS, AND VITS

Categories	Computation Layers
CNN-specific Layers	Convolutional (Conv) Layer
GNN-specific Layers	Message Passing (MP) Layer Vector Inner Product (VIP) Layer
ViT-specific Layers	Multi-head Self-attention (MSA) Layer
Common Layers	Softmax Layer Activation Layer Linear Layer Normalization Layer
Data Manipulation Layers	Matrix Transpose Layer

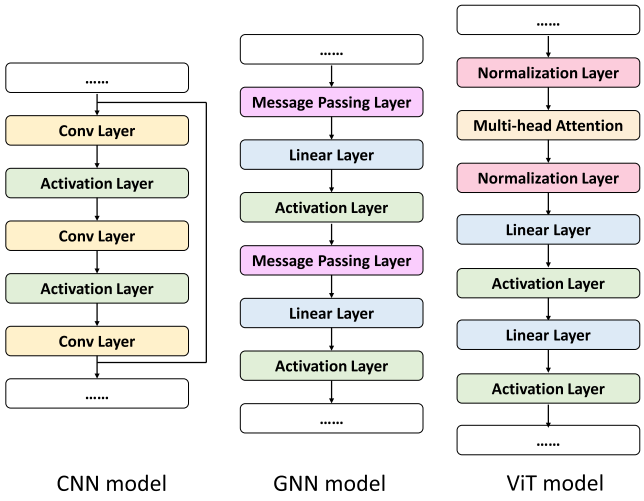


Fig. 4. Examples of using the IR to represent the computation graphs (task dependency graphs) of CNN, GNN, and ViT.

and (3) the proposed bytecode that connects the compiler and the runtime system.

A. Intermediate Representation

We identify the essential computation layers in CNNs, GNNs, and ViTs, as summarized in Table I. These layers are divided into several categories. These computation layers can represent a broad range of models, and several examples are shown in Fig. 4. The intermediate representation (IR) of a computation layer stores the key parameters of this layer. For an input model, the compiler first translates it into an intermediate representation that represents the computation graph (i.e., task dependency graph) of the inference process. We use TQ to denote the task dependency graph represented by the IR for the input model. In TQ, each node represents a computation layer, and the edges indicate the data dependencies between nodes (layers). See the examples in Fig. 4. We introduce these computation layers as follows:

Convolutional (Conv) Layer: Conv Layer is a CNN-specific layer and is essential in CNNs. As shown in Fig. 3, the input to a Conv layer is indicated as $\mathfrak{F}_{\text{in}}$ that has c_{in} feature maps, each having a size of $h_{\text{in}} \times w_{\text{in}}$. The output of a Conv layer is denoted as $\mathfrak{F}_{\text{out}}$ which has c_{out} feature maps with each having the size of $h_{\text{out}} \times w_{\text{out}}$. The convolution kernel $\mathfrak{W}$ has the size of $c_{\text{out}} \times c_{\text{in}} \times k_1 \times k_2$. A Conv Layer can be expressed as: $\mathfrak{F}_{\text{out}}[j] = \sum_{g=0}^{c_{\text{in}}-1} \mathfrak{W}[j,g] \star \mathfrak{F}_{\text{in}}[g]$, where $\mathfrak{F}[g]$ denotes the gth feature map of $\mathfrak{F}$, $\mathfrak{W}[j,g] \in \mathbb{R}^{k_1 \times k_2}$ denotes the kernel matrix corresponding to $\mathfrak{F}_{\text{out}}[j]$ and $\mathfrak{F}_{\text{in}}[k]$, and $\star$ is the 2D convolution operator.

Message Passing (MP) Layer: A message passing layer is used in GNNs for message passing within graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. The input are vertex feature vectors $\{\mathbf{h}_{\text{in}}[v] \in \mathbb{R}^f : v \in \mathcal{V}\}$ and edges $\{e_{vu} : e_{vu} \in \mathcal{E}\}$. The output vertex feature vectors $\{\mathbf{h}_{\text{out}}[v] \in \mathbb{R}^f : v \in \mathcal{V}\}$ are obtained through message passing:

$$\mathbf{h}_{\text{out}}[v] = \rho(\{e_{uv} \cdot \mathbf{h}_{\text{in}}[u] : u \in \mathcal{N}(v)\}) \tag{1}$$

where $\mathcal{N}(v)$ denotes the set of neighbors of v, and $\rho()$ is the element-wise reduction function, such as Max() and Sum().

Vector Inner Product (VIP) Layer: In many GNNs (e.g., GAT [6]), the inner product of two vertex feature vectors is used to calculate the edge weight. The inputs are the vertex feature vectors $\{\mathbf{h}_{\text{in}}[v] \in \mathbb{R}^f : v \in \mathcal{V}\}$, and predefined edge connectivity $\{e_{vu} : e_{vu} \in \mathcal{E}\}$ with the value of e_{vu} to be calculated. The calculation of e_{vu} can be expressed as: $e_{uv} = \langle \mathbf{h}_{\text{in}}[u], \mathbf{h}_{\text{in}}[v] \rangle$ where $\langle , \rangle$ denotes vector inner product.

Multi-head Self-attention (MSA) Layer: Multi-head self-attention (MSA) layer is the basic building block in ViTs. The input to the MSA layer is a feature matrix $\mathbf{H}_{\text{in}} \in \mathbb{R}^{N \times D}$ with each row being a vector token. N is the number of input tokens and D is the length of each token. MSA introduces the concept of query, key, and value, abbreviated as $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$, respectively. The computation process of an MSA layer can be represented as:

$$\begin{aligned}
&\mathbf{Q}_i = \mathbf{H}_{\text{in}} \times \mathbf{W}_i^{\mathbf{Q}}; \mathbf{K}_i = \mathbf{H}_{\text{in}} \times \mathbf{W}_i^{\mathbf{K}}; \mathbf{V}_i = \mathbf{H}_{\text{in}} \times \mathbf{W}_i^{\mathbf{V}};\\
&\mathbf{P}_i = \text{softmax}(\mathbf{Q}_i \times \mathbf{K}_i^{\top} / \sqrt{h});\\
&\mathbf{G}_i = \mathbf{P}_i \times \mathbf{V}_i;\\
&\mathbf{H}_{\text{out}} = [\mathbf{G}_1; \mathbf{G}_2; \ldots; \mathbf{G}_k] \times \mathbf{W}_{\text{msa}};\\
&\text{where } \mathbf{W}_i^{\mathbf{Q}}, \mathbf{W}_i^{\mathbf{K}}, \mathbf{W}_i^{\mathbf{K}} \in \mathbb{R}^{D \times h}, \text{ and } \mathbf{W}_{\text{msa}} \in \mathbb{R}^{k \cdot h \times D}
\end{aligned} \tag{2}$$

where k $(1 \leqslant i \leqslant k)$ is the number of attention head.

$\mathbf{Q}_i$, $\mathbf{K}_i$, and $\mathbf{V}_i$ are the query matrix, key matrix, and value matrix of ith head, respectively. h is the hidden dimension, $\mathbf{P}_i$ is the attention score matrix of ith head, $\mathbf{G}_i$ is the aggregated values calculated through applying the attention score, and $\mathbf{H}_{\text{out}} \in \mathbb{R}^{N \times D}$ is the output of the MSA layer. $\{\mathbf{W}_i^{\mathbf{Q}}, \mathbf{W}_i^{\mathbf{K}}, \mathbf{W}_i^{\mathbf{V}} : (1 \leqslant i \leqslant k)\}$, and $\mathbf{W}_{\text{msa}}$ are the weight matrices.

Common Layers: There is a set of commonly used layers in CNNs, GNNs, and ViTs, such as softmax layer, element-wise activation layer, linear layer, and normalization layer (e.g., layer normalization, batch normalization).

Data Manipulation (DM) Layers: A data manipulation layer defines the necessary data manipulation operations required to be performed between two computation layers, such as matrix transpose.

TABLE II
SET OF MATRIX OPERATIONS

Name	Notation	Name	Notation
Matrix Multiplication	$\mathbf{X} \times \mathbf{Y}$	Scalar-Matrix Multiplication	$\alpha\mathbf{X}$
Matrix Addition	$\mathbf{X} + \mathbf{Y}$	Matrix Reduction	For input $\mathbf{X}$, performs reduction in a dimension
Element-Wise Function	$\sigma(\mathbf{X})$	Sampled Dense Matrix-Matrix Multiplication	$\mathbf{S} \odot (\mathbf{X} \times \mathbf{Y})$
Matrix-Vector Multiplication	$\mathbf{X} \times \mathbf{V}$		

B. Step 1: Layer Fusion

Several types of layers can be combined with adjacent layers to facilitate task-level parallelism, reduce external memory traffic, and reduce overall computational complexity. This leads to the proposed *layer fusion optimization*. (1) *Activation Fusion:* An activation layer can be merged with its preceding layer (e.g., the Convolutional layer). The element-wise activation operation can be executed immediately after its preceding layer without reloading the preceding layer's output. (2) *Norm Fusion:* The coefficients are fixed during inference in the element-wise normalization operation. Hence, the coefficients of a Norm layer can be merged with the weights and biases of its adjacent convolutional or linear transformation layer. This reduces external memory traffic and overall computational complexity. (3) *Data Manipulation Fusion:* A Data Manipulation (DM) layer can be combined with its subsequent computation layer. Consequently, the data manipulation operations and the computations of the following layer can be pipelined, enabling task-level parallelism.

C. Step 2: Mapping Computation Layers to Matrix Operations

Step 2 maps each computation layer to matrix operations (Table II) that are associated with basic computation primitives supported by the accelerator.

Mapping Conv Layer: We employ the kn2row algorithm [46] to map a Conv Layer into matrix operations (See Fig. 5). The convolution kernel matrix $\mathfrak{W}$ is rearranged into $k_1 \times k_2$ submatrices, denoted as $\{\mathbf{M}_{\text{KE}(i)} : 0 \leqslant i \leqslant k_1 k_2 - 1\}$, where each $\mathbf{M}_{\text{KE}(i)}$ has dimensions $c_{\text{in}} \times c_{\text{out}}$. The input feature maps $\mathfrak{F}_{\text{in}}$ are organized into a matrix denoted $\mathbf{M}_{\text{IF}}$ of size $c_{\text{in}} \times h_{\text{in}} w_{\text{in}}$, with each input feature map represented as a row in this matrix. Each $\mathbf{M}_{\text{KE}(i)}$ is multiplied by $\mathbf{M}_{\text{IF}}$ to obtain $k_1 k_2$ output matrices, denoted as $\{\mathbf{M}_{\text{OF}(i)} : 0 \leqslant i \leqslant k_1 k_2 - 1\}$, each having dimensions $c_{\text{out}} \times h_{\text{out}} w_{\text{out}}$. Through shift and add (shift-add) operations, the $k_1 k_2$ output matrices are merged into a single output matrix $\mathbf{M}_{\text{OF}}$ of size $c_{\text{out}} \times h_{\text{out}} w_{\text{out}}$. $\mathbf{M}_{\text{OF}}$ can be further reorganized back to c_{out} output feature maps. Consequently, a Conv Layer is mapped to matrix multiplication and matrix addition operations.

Discussion on the mapping of Conv Layer: The proposed mapping strategy brings several benefits: (1) The computation of a Conv Layer can be easily mapped to the matrix operations associated with the computation primitives. (2) The reorganization of data layout for the kernel matrix ($\mathfrak{W}$) occurs at compile time, incurring a one-time cost. This eliminates the need for data layout transformations at runtime. (3) The data layout for both the input feature maps and the output feature maps remains consistent without the need for data layout transformations between consecutive Conv Layers. (4) Most importantly, the data layout of the input/output feature maps can accommodate the data layout of GNNs and ViTs in computer vision tasks. For example, in GNN-based image segmentation [47], each input feature map is treated as a vertex, which corresponds to a row in matrix $\mathbf{M}_{\text{IF}}/\mathbf{M}_{\text{OF}}$ (Fig. 5). Moreover, in many GNNs or ViTs, a single pixel or a patch of pixels in the feature maps forms a feature vector, corresponding to single or multiple columns in matrix $\mathbf{M}_{\text{IF}}/\mathbf{M}_{\text{OF}}$. As a result, the computation layers of GNNs or ViTs can directly fetch the vertex feature vectors from $\mathbf{M}_{\text{IF}}/\mathbf{M}_{\text{OF}}$ or transposed version of $\mathbf{M}_{\text{IF}}/\mathbf{M}_{\text{OF}}$, without complex data layout transformation. For example, the well-known im2col [48] mapping algorithm needs to duplicate the features and reorganize the feature maps based on the kernel matrix $\mathfrak{W}$, which may lead to a large overhead.

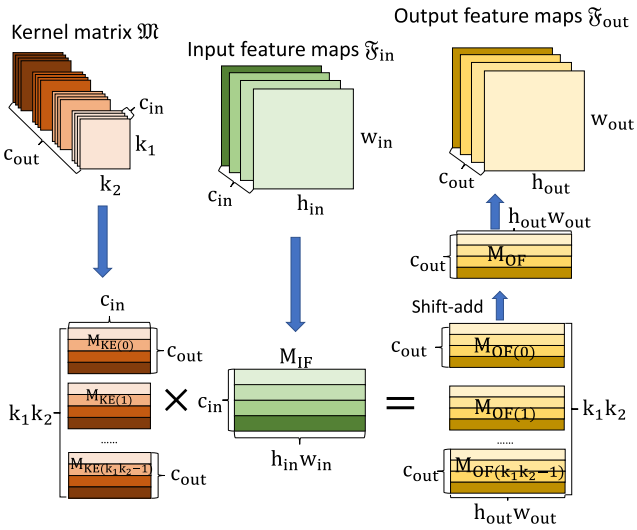


Fig. 5. Mapping a convolutional layer to matrix operations.

Mapping Messaging Passing Layer: A messaging passing layer operates on an input graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, and is assigned to matrix multiplication $\mathbf{A} \times \mathbf{H}$, where $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ denotes the graph adjacency matrix, and $\mathbf{H} \in \mathbb{R}^{|\mathcal{V}| \times f}$ represents the feature matrix which is the concatenation of vertex feature vectors.

Mapping Vector Inner Product (VIP) Layer: A vector inner product (VIP) layer can be mapped to the sampled dense-dense matrix multiplication. For a VIP layer, suppose the adjacency matrix of the input graph is $\mathbf{A}$ and the concatenation of the input vertex feature vectors $\{\mathbf{h}_{\text{in}}[v] \in \mathbb{R}^f : v \in \mathcal{V}\}$ is $\mathbf{H}_{\text{in}}$. The output feature matrix is calculated by $\mathbf{H}_{\text{out}} = \mathbf{A} \odot (\mathbf{H}_{\text{in}} \times \mathbf{H}_{\text{in}}^{\top})$, where $\odot$ is the element-wise multiplication.

Mapping Multi-head Self-attention Layer: The computation process of a multi-head self-attention layer is demonstrated in (2). The processing of calculating queries $\mathbf{Q}_i (1 \leqslant i \leqslant k)$, keys $\mathbf{K}_i (1 \leqslant i \leqslant k)$, and values $\mathbf{V}_i (1 \leqslant i \leqslant k)$ are mapped to matrix multiplications. The process of calculating attention scores $\mathbf{P}_i (1 \leqslant i \leqslant k)$ is mapped to matrix multiplication, scalar-matrix

Algorithm 1: Mapping Computation Layers to Matrix Operations.

Input: Task dependency graph TQ of the input model with each node represents a computation layer;
Output: Task dependency graph TQ of the input model with each node represents a set of matrix operations associated with a computation layer.
1: **for** each computation layer l in TQ **do**
2: Map layer l to matrix operations

multiplication, matrix reduction, and matrix-vector multiplication. Moreover, the process of calculating $\mathbf{G}_i (1 \leqslant i \leqslant k)$ and $\mathbf{H}_{\text{out}}$ is mapped to matrix multiplication.

Mapping Common Layers: Similarly, the common layers can be mapped to the matrix operations in Table II.

D. Step 3: Data Tiling and Sparsity Preprocessing

Through step 2, in the computation graph (task dependency graph), each computation layer is mapped to a set of matrix operations. A matrix operation generated in step 2 is denoted as $\mathcal{M}(\mathbf{Z} = q(\mathbf{X}, \mathbf{Y}))$, where $\mathbf{X}$ and $\mathbf{Y}$ are the input matrices, $\mathbf{Z}$ is the output matrices, and $q(\cdot)$ denotes the operator (e.g., matrix multiplication, matrix addition). Due to the limited size of on-chip buffers, data tiling is required. Since each computation layer is mapped to matrix operations (Table II), the compiler performs two-dimensional (2-D) block data partitioning for each matrix operation. Therefore, each large matrix operation is decomposed into a set of matrix operations of smaller size. Suppose the on-chip memory size is $N \times N$. For example, for a matrix multiplication $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$, we perform 2-D block partitioning for $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ such that each output partition $\mathbf{Z}_{ij} \in \mathbb{R}^{N \times N}$ is calculated by $\mathbf{Z}_{ij} = \sum \mathbf{X}_{ik} \times \mathbf{Y}_{kj}$. We define the workload for calculating an output partition $T(\mathbf{Z}_{ij})$ as a *computation task*. Therefore, a matrix operation is partitioned into a set of independent computation tasks. As these output partitions have no data dependence between each other, these computation tasks can executed in parallel. For the data that are already known at compile time (e.g., weight matrices, input graph adjacency matrix), the compiler profiles their data sparsity. For the intermediate results that are unknown at compile time, the compiler puts *placeholders* for its data sparsity. The accelerator will profile their data sparsity at runtime and send the sparsity information to the runtime system. The runtime system uses the data sparsity information to perform dynamic sparsity exploitation.

E. Step 4: Bytecode Generation

Unlike the existing compilers (e.g., GraphAGILE [22] and OPU [20]) for ML accelerators which directly generate hardware instructions, VisionAGILE's compiler produces a sequence of bytecodes. Table III shows the bytecodes that represent matrix operations. Note that the hardware accelerator does not directly execute these bytecodes; instead, the runtime system performs just-in-time (JIT) compilation to generate hardware instructions from the bytecodes dynamically. A bytecode defines a matrix operation, including the metadata of the matrix operation and input matrices. These metadata consists of data sparsity, data layout, and data format of input matrices. These properties can be determined at compile time or runtime, depending on their availability. This flexibility in the bytecode enables the implementation of our proposed runtime optimizations: (1) *Dynamic sparsity exploitation:* First, bytecode generation does not rely on data sparsity, layout, or format information. This enables the runtime system to perform dynamic sparsity exploitation that maps matrix operations to hardware primitives dynamically. Since the data sparsity of intermediate results is unknown during compilation, the compiler does not need to deal with the data sparsity information for the intermediate results. The hardware accelerator will profile the data sparsity for intermediates results and annotate it to the bytecode at runtime. (2) *Determining data layout and data sparsity dynamically at runtime:* As data sparsity information for intermediate results remains unknown at compile time, the corresponding data layout and data format are also unknown. Using bytecode allows the runtime system to dynamically determine the data layout and format for the input matrices. (3) *Dynamic task scheduling:* Utilizing bytecode also facilitates dynamic task scheduling by the runtime system for load balance.

Algorithm 2: Data Tiling and Sparsity Preprocessing.

Input: Task dependency graph TQ of the input model with each matrix operation unpartitioned;
Output: Task dependency graph TQ of the input model with each matrix operation partitioned.
1: **for** each matrix operation $\mathcal{M}(\mathbf{Z} = q(\mathbf{X}, \mathbf{Y}))$ in TQ **do**
2: $\mathbf{Z}$ has the dimension of $d_1 \times d_2$
3: Perform 2-D block data partitioning for $\mathbf{Z}, \mathbf{X}, \mathbf{Y}$
4: Obtain a set of computation tasks $\{T(\mathbf{Z}_{ij}) : 1 \leqslant i \leqslant \frac{d_1}{N}, 1 \leqslant j \leqslant \frac{d_2}{N}\}$ for calculating $\mathcal{M}(\mathbf{Z} = q(\mathbf{X}, \mathbf{Y}))$, where $T(\mathbf{Z}_{ij})$ denotes the computation task for calculating $\mathbf{Z}_{ij}$
5: Profile the data sparsity of the data blocks that are known at compile time

V. Hardware Design

This Section introduces the hardware architecture design, including the key computation primitives supported by accelerator (Section V-A), and the microarchitecture design.

A. Computation Primitives

The overview of the hardware platform is shown in Fig. 2. The hardware accelerator of VisionAGILE consists of a controller and multiple parallel processing elements (PEs). A PE can execute a set of computation primitives associated with the matrix operations. Note that a matrix multiplication operation can be mapped to various hardware primitives based on the data sparsity. For other matrix operations, each of them corresponds to one hardware primitive. The mapping

TABLE III
BYTECODE OF VISIONAGILE

Category	Description	Bytecode
Memory Operations	Memory Load	`Load [] Buffer [] Layout [] Format [] Sparsity []`
	Memory Store	`Store [] Buffer [] Layout [] Format [] Sparsity []`
Computation Operations	Matrix Multiplication (MM)	`MM [X][Y] SparsityX [] SparsityY []`
	Matrix Addition (MA)	`MA [X][Y]`
	Element-wise Function (EF)	`EF [X] Function []`
	Matrix Vector Multiplication (MV)	`MV [X][V]`
	Scalar-Matrix Multiplication (ScalarM)	`ScalarM [$\alpha$] [X]`
	Matrix Reduction (MR)	`MR [X] Dimension []`
	Sampled Dense-Dense Matrix Multiplication (SDDMM)	`SDDMM [S][X][Y]`

[] represents a variable that will be determined at compile time or at runtime.

TABLE IV
MAPPING BETWEEN MATRIX OPERATIONS AND COMPUTATION PRIMITIVES

Matrix Operations	Computation Primitives
Matrix Multiplication (MM)	DDMM (Dense-dense MM) SpDMM (Sparse-dense MM) SPMM (Sparse-sparse MM)
Matrix Addition (MA)	MatAdd
Element-Wise Function (EF)	MatEF
Matrix-Vector Multiplication (MV)	MVMat
Scalar-Matrix Multiplication (ScalarM)	SMMat
Matrix Reduction (MR)	MatRedu
Sampled Dense Matrix-Matrix Multiplication (SDDMM)	SDDMM

between matrix operations and hardware primitives is elaborated in Table IV. Note that a matrix multiplication operation can be mapped to three hardware primitives: Dense-dense matrix multiplication (DDMM), Sparse-dense matrix multiplication (SpDMM), and Sparse-sparse matrix multiplication (SPMM).

The proposed mapping strategy for matrix multiplication enables the runtime system to perform dynamic sparsity exploitation that dynamically maps the matrix multiplication operation to DDMM, SpDMM, or SPMM based on the data sparsity. This can potentially reduce the computational complexity, leading to reduced inference latency.

B. Data Format and Data Layout

We define the *data format* and *data layout* that are used by various computation primitives. *Data format:* We store the input matrices for a primitive using *sparse* format or *dense* format. We use the coordinate (COO) format to represent a sparse matrix where a nonzero element is represented using a three-tuple (col, row, val) denoting the column index, row index, and value, respectively. The COO format is the standard data format used in state-of-the-art GNN libraries [49]. *Data layout:* Data layout defines the order of storing the matrix elements. For a sparse matrix in row-major order, elements within the same row are stored in contiguous memory locations. Otherwise, it is in column-major order. Similarly, the row-major and column-major order for a dense matrix can be derived.

Notations: For a matrix $\mathbf{X}$, we use $\mathbf{X}[i]$ to denote the ith row of $\mathbf{X}$ and use $\mathbf{X}[i:j]$ to denote the submatrix of $\mathbf{X}$ from ith row to $(j-1)$th row. We use $\mathbf{X}[i][j]$ to denote the element of $\mathbf{X}$ at the ith row and the jth column. An element (j, i, val) in sparse matrix $\mathbf{X}$ is also denoted as $\mathbf{X}[i][j] = val$.

C. Microarchitecture

The microarchitecture of the processing element (PE) is depicted in Fig. 6. The PE has a flexible data path and memory organization designed to execute various computation primitives. The PE offers multiple execution modes, each corresponding to a specific hardware primitive. The PE is equipped with hardware multiplexers to select its execution mode, and switching between these modes incurs a one-clock-cycle overhead. The PE includes a two-dimensional Arithmetic Logic Unit (ALU) array with the dimension of $p_{aa} \times p_{aa}$. Each ALU can perform basic arithmetic and logic operations, such as multiplication and addition. Additionally, there are several data buffers for storing input and output data, including Scalar Buffer (SB), Vector Buffer 1 (VB1), Vector Buffer 2 (VB2), and Result Buffer (RB). Each data buffer has p_{aa} memory banks to enable parallel on-chip memory access. There are all-to-all data communication networks – a Buffer-to-Buffer (B2B) routing network and a Buffer-to-pipeline routing network. Furthermore, each PE incorporates an Instruction Queue (IQ) that receives incoming instructions generated by the runtime system and an Instruction Decoder (ID) responsible for decoding the instructions for hardware execution. Moreover, for dynamic sparsity exploitation, the PE incorporates a Data Layout Management Unit (DLM) and a Data Format Management Unit (DFM) for Data Layout and format transformation during runtime. The Sparsity Profiler profiles the data sparsity during runtime and sends the data sparsity information to the soft processor. We introduce the various execution modes of the PE as follows:

DDMM mode: It executes matrix multiplication $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$ and takes both two input matrices as dense matrices. To this end, the ALU array is organized as a two-dimensional systolic array (See Fig. 6). The ALU array executes matrix multiplication using output stationary dataflow and can execute p_{aa}^2 multiply accumulation (MAC) operations per clock cycle.

SpDMM Mode: It executes matrix multiplication $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$, and takes $\mathbf{X}$ as sparse matrix and $\mathbf{Y}$ as dense matrix. To this end, the ALU array is divided into $p_{aa}/2$ Scatter Units (SCU) and $p_{aa}/2$ Gather Units (GAU). Each Scatter Unit or Gather Unit has an ALU array of size $p_{aa}/2 \times 2$. SpDMM is executed using the Scatter-Gather Paradigm as shown in Algorithm 3. The sparse matrix $\mathbf{X}$ (in Scalar Buffer) is stored in row-major order using

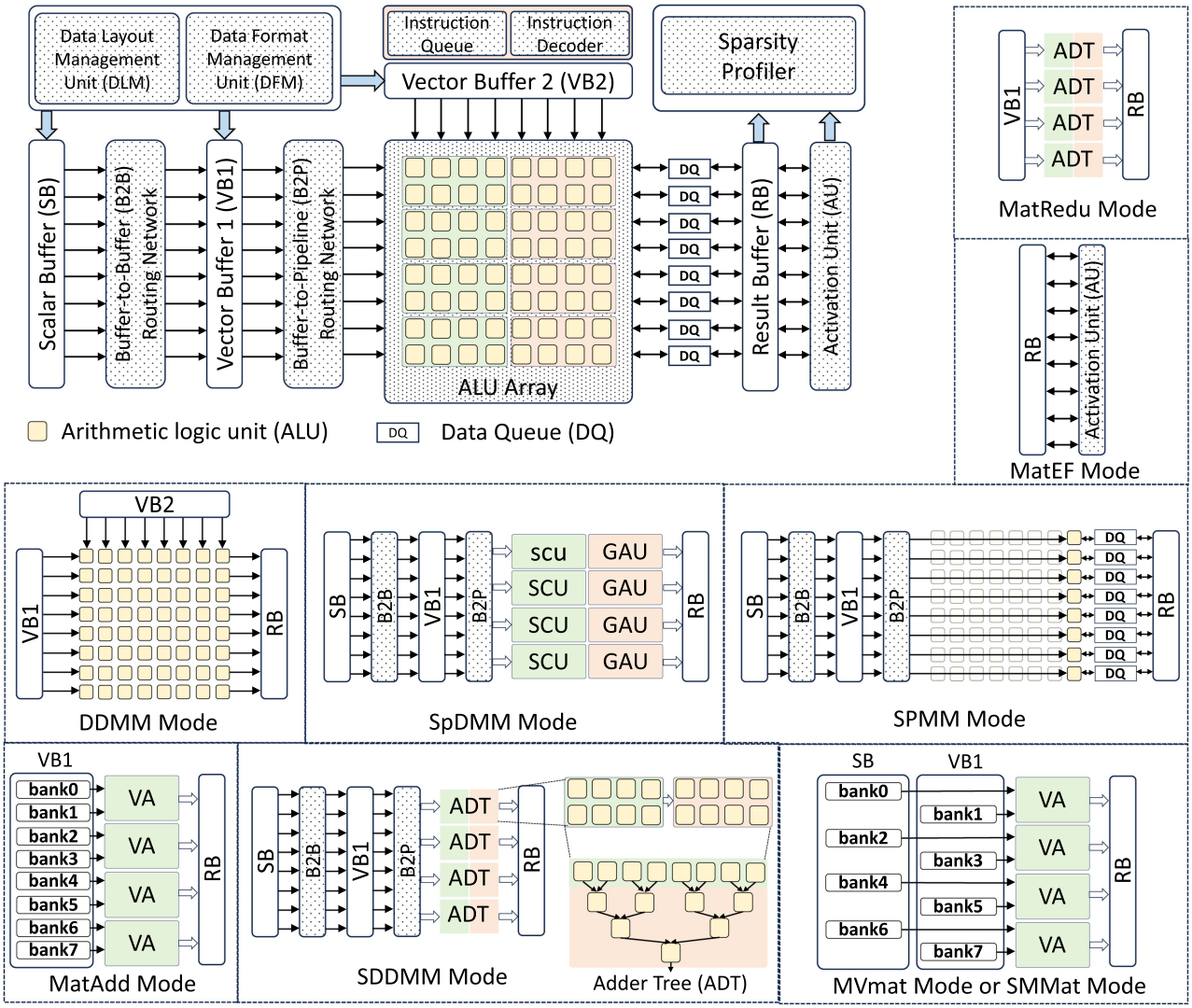


Fig. 6. Architecture of the processing element and the supported execution modes each corresponds to a computation primitive.

COO format. The dense matrix $\mathbf{Y}$ is stored in row-major order using the dense format, and $\mathbf{Y}[i]$ is stored in the bank $(i \bmod p_{aa})$ of Vector Buffer 1. Each non-zero element $e(i, j, val)$ in $\mathbf{X}$ is fetched from the Scalar Buffer ($p_{aa}/2$ elements can be fetched from Scalar Buffer per clock cycle) and sent to the B2B network. Then, e is routed to bank $(i \bmod p_{aa})$ for fetching $\mathbf{Y}[i]$, which forms the input data pair $(\mathbf{Y}[i], e)$. The input pair is routed to the $(j \bmod p_{aa}/2)$th SCU and GAU. The Scatter Unit (SCU) performs the multiplication of $e.val$ and $\mathbf{Y}[i]$ to produce the intermediate result u. Then, the corresponding Gather Unit adds u to $\mathbf{Z}[j]$. Unlike the DDMM mode, SpDMM mode can skip the computation for the zero elements in matrix $\mathbf{X}$. It can reduce the computation complexity when there are zero elements in $\mathbf{X}$. SpDMM mode can execute p_{aa}^2 MAC operations per clock cycle.

SPMM mode: It executes matrix multiplication $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$ and takes both $\mathbf{X}$ and $\mathbf{Y}$ as sparse matrices. The ALU array is organized as p_{aa} parallel Sparse Computation pipelines (SCP). Each SCP has two ALUs to perform the multiplication of two non-zero elements and the merging of intermediate results. Each SCP has a Data Queue (DQ) to store the intermediate results in sparse format. The multiplication of two input sparse matrices is executed using Row-wise Product with Scatter-Gather paradigm as shown in Algorithm 4. For Row-wise Product, a row $\mathbf{Z}[j]$ of

Algorithm 3: SpDMM using Scatter-Gather Paradigm.

Input: Sparse matrix (Scalar Buffer): $\boldsymbol{X}$; Dense matrix (Vector Buffer 1): $\boldsymbol{Y}$;
Output: Output matrix (Result Buffer): $\boldsymbol{Z} = \boldsymbol{X} \times \boldsymbol{Y}$;

```
1: while not done do
2:   for each e(i, j, value) in X Parallel do ▷Scatter Phase
3:     B2B routes e to VB1
4:     Fetch Y[i] from VB1
5:     Form input pair (Y[i], e)
6:     B2P routes input pair to (j mod p_aa)th SCU and GAU
7:   for each input pair Parallel do ▷ Gather Phase
8:     u ←Scatter(Y[i], e.value) ▷ Scatter Unit (SCU)
9:     Fetch Z[j] from Result Buffer
10:    Z[j] ← Gather(u) ▷Gather Unit (GAU)
```

Algorithm 4: SPMM Using Row-Wise Product With Scatter-Gather Paradigm.

Input: Sparse matrix (Scalar Buffer): $\mathbf{X}$; Sparse matrix (Vector Buffer 1): $\mathbf{Y}$;
Output: Output matrix (Result Buffer): $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$;

1: **for** each row $\mathbf{Z}[j]$ in $\mathbf{Z}$ **Parallel do**
2: Assign the workload of $\mathbf{Z}[j]$ to $(j \bmod p_{aa})$th SCP
3: load $\mathbf{Z}[j]$ to the Data Queue from Results Buffer
4: **for** each $e(i, j, value)$ in $\mathbf{X}[j]$ **do** ▷ Scatter Phase
5: B2B routes e to VB1
6: Fetch $\mathbf{Y}[i]$ from VB1
7: Form input pair ($\mathbf{Y}[i]$, e)
8: B2P routes input pair to $(j \bmod p_{aa})$th SCP
9: **for** each input pair ($\mathbf{Y}[i]$, e) **do** ▷ Gather Phase
10: **for** each non-zero $\mathbf{Y}[i][k]$ in $\mathbf{Y}[i]$ **do** ▷ SCP
11: Produce $u \leftarrow e.value \times \mathbf{Y}[i][k]$
12: Accumulate $\mathbf{Z}[j][k] \leftarrow u$
13: Store $\boldsymbol{Z}[j]$ to the Result Buffer ▷ Obtain $\boldsymbol{Z}[j]$

output matrix $\mathbf{Z}$ is calculated through:

$$\boldsymbol{Z}[j] = \sum_i \boldsymbol{X}[j][i] * \boldsymbol{Y}[i] \tag{3}$$

For calculating the output matrix $\mathbf{Z}$, an SCP is assigned the workload of a row of output matrix. p_{aa} SCPs can calculate p_{aa} output rows in parallel until all the rows of output matrices are calculated. To efficiently execute a Row-wise Product, all input sparse matrices ($\mathbf{X}$, $\mathbf{Y}$) and output matrices $\mathbf{Y}$ are stored using COO format in row-major order. Compared with the DDMM mode, the SPMM mode can skip the computation for the zero-elements in $\mathbf{X}$ and $\mathbf{Y}$. SPMM mode can execute p_{aa} multiply-accumulate (MAC) operations per clock cycle.

MatAdd mode: It executes matrix addition $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$. The two input matrices are stored in Vector Buffer 1, where $\mathbf{X}$ is stored in banks $i\,(0 \leqslant i \leqslant p_{aa}, (i \bmod 2) == 0)$ and $\mathbf{Y}$ is stored in banks $j\ (0 \leqslant j \leqslant p_{aa}, (j \bmod 2) == 1)$. Half of the ALU array is organized as $p_{aa}/2$ vector adders (VA) with each having p_{aa} ALUs to perform addition. The $p_{aa}/2$ vector adders can execute $p_{aa}^2/2$ addition operations per clock cycle.

SDDMM mode: It executes sampled dense-dense matrix multiplication $\mathbf{Z} = \mathbf{S} \odot (\mathbf{H} \times \mathbf{H}^\top)$. $\mathbf{S}$ is the sampling matrix stored in Scalar Buffer using sparse format. $\mathbf{H}$ is a dense matrix stored in Vector Buffer 1 using dense format and row-major order. The ALU array is organized as $p_{aa}/2$ adder trees (ADT). Each adder tree performs the vector inner product of two vectors. The execution of SDDMM follows the Scatter-Gather Paradigm as shown in Algorithm 5. For a non-zero element $e(i, j)$ of $\mathbf{S}$ from bank $k\,(0 \leqslant k \leqslant p_{aa} - 1)$ of Scalar Buffer, the indices i and j are routed to Vector Buffer 1 separately through the B2B network for fetching $\mathbf{H}[i]$ and $\mathbf{H}_j$. Then, $\mathbf{H}[i]$ and $\mathbf{H}[j]$ are routed to the kth adder tree through the B2P network for vector inner product. The partial result $\langle \mathbf{H}[i], \mathbf{H}[j] \rangle$ is accumulated to

Algorithm 5: SDDMM Using Scatter-Gather Paradigm.

Input: Sampling matrix (Scalar Buffer): $\mathbf{S}$; Dense matrix (Vector Buffer 1): $\mathbf{H}$;
Output: Output matrix (Result Buffer): $\mathbf{Z} = \mathbf{S} \odot (\mathbf{H} \times \mathbf{H}^\top)$;

1: **for** each non-zero $e(i, j)$ in $\mathbf{S}$ **Parallel do** ▷ Scatter Phase
2: Suppose $e(i, j)$ is from bank k of Scalar Buffer
3: B2B routes indices i and j to VB1
4: Fetch $\mathbf{H}[i]$ and $\mathbf{H}[j]$ from VB1
5: B2P routes $\mathbf{H}[i]$ and $\mathbf{H}[j]$ to $(k \bmod p_{aa})$th ADT
6: **for** each input pair ($\mathbf{H}[i]$, $\mathbf{H}[j]$) **do** ▷ Gather Phase
7: Produce $u \leftarrow \langle \mathbf{H}[i], \mathbf{H}[j] \rangle$
8: Accumulate $\mathbf{Z}[i][j] \leftarrow u$

$\mathbf{Z}[i][j]$. SDDMM mode can execute $p_{aa}^2/2$ MAC operations per clock cycle.

MVMat mode and SMMat mode: MVMat mode executes matrix-vector multiplication $\mathbf{Z} = \mathbf{X} \times \mathbf{V}$ and SMMat mode executes scalar-matrix multiplication $\mathbf{Z} = \alpha\mathbf{X}$. Both matrix-vector multiplication and scalar-matrix multiplication can be decomposed into a set of scalar-vector multiplication. Therefore, these two execution modes share the same data path as shown in Fig. 6. The input scalars are stored in the Scalar Buffer and the input vectors are stored in Vector Buffer 1. The ALU array is organized into multiple Vector Multipliers (VM) and multiple Vector Accumulators (VACC), with each having p_{aa} ALUs. MVMat or SMMat can execute $p_{aa}^2/2$ MAC operations per clock cycle.

MatRedu mode: For an input matrix $\mathbf{X}$, MatRedu performs reduction/accumulation in one dimension of $\mathbf{X}$, which can be used to execute the softmax activation function.

MatEF mode: It utilizes the Activation Unit (AU) and performs the element-wise activation for the input $\mathbf{X}$.

1) Sparsity Profiler: The sparsity profiler is composed of an array of hardware comparators that compare each output element to zero, counting the total number of zeros in the output matrices. After the entire output matrix is processed, the profiler then uses a hardware divider to calculate the sparsity of the output matrix by dividing the total number of zeros by the total number of elements. The sparsity of the matrix $\mathbf{X}$ is denoted as $\beta(\mathbf{X})$ (see Section VII-A).

D. Data Layout and Data Format Management

As summarized in Table V, different computation primitives require different data layouts and formats for input and output data.

To this end, we develop the hardware mechanisms for transforming data layout and format, which are implemented in the Data Layout Management (DLM) Unit and Data Format Management (DFM) Unit.

Data Layout Management (DLM) Unit: DLM Unit transforms the data layout between row-major order and column-major order for the data stored in dense format, equivalent to matrix transpose. The process of matrix transpose and the proposed

TABLE V
DATA BUFFER, DATA LAYOUT AND DATA FORMAT REQUIRED BY VARIOUS HARDWARE PRIMITIVES

Hardware Primitives	Data		Buffer	Data Layout	Data Format
DDMM $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$	Input	$\mathbf{X}$	VB1	Row-major	Dense
		$\mathbf{Y}$	VB2	Column-major	Dense
	Output	$\mathbf{Z}$	RB	Row-major	Dense
SpDMM $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$	Input	$\mathbf{X}$	SB	[Row/column]-major	Sparse
		$\mathbf{Y}$	VB1	Row-major	Dense
	Output	$\mathbf{Z}$	RB	Row-major	Dense
SPMM $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$	Input	$\mathbf{X}$	SB	Row-major	Sparse
		$\mathbf{Y}$	VB1	Row-major	Sparse
	Output	$\mathbf{Z}$	RB	Row-major	Dense
SDDMM $\mathbf{Z} = \mathbf{S} \odot (\mathbf{H} \times \mathbf{H}^\top)$	Input	$\mathbf{S}$	SB	[Row/column]-major	Sparse
		$\mathbf{H}$	VB1	Row-major	Dense
	Output	$\mathbf{Z}$	RB	[Row/column]-major	Sparse
MVMat $\mathbf{Z} = \mathbf{X} \times \mathbf{V}$	Input	$\mathbf{X}$	VB1	Row-major	Dense
		$\mathbf{V}$	SB	[Row/column]-major	Sparse
	Output	$\mathbf{Z}$	RB	Row-major	Dense
SMMat $\mathbf{Z} = \alpha\mathbf{V}$	Input	α	SB	[Row/column]-major	Sparse
		$\mathbf{X}$	VB1	Row-major	Dense
	Output	$\mathbf{Z}$	RB	Row-major	Dense
MatRedu	Input	$\mathbf{X}$	VB1	Row-major	Dense
	Output	$\mathbf{Z}$	RB	Row-major	Dense
MatEF $\mathbf{Z} = \sigma(\mathbf{X})$	Input	$\mathbf{X}$	VB1	Row-major	Dense
	Output	$\mathbf{Z}$	RB	Row-major	Dense

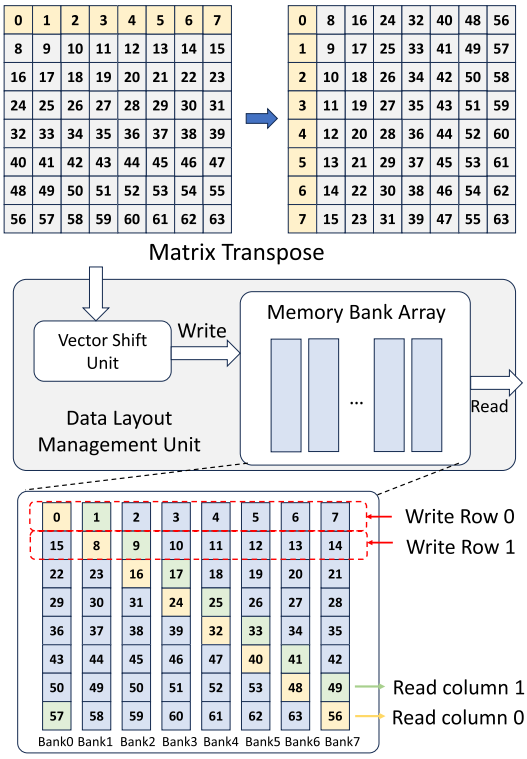


Fig. 7. Data layout transformation process and the proposed Data Layout Management Unit.

DLM Unit are illustrated in Fig. 7. In the DLM Unit, there is a Vector Shift Unit, which performs the circular shift for the rows of the input matrices. Then, the Vector Shift Unit writes the input rows to the Memory Bank Array (MBA). The stored vectors are read diagonally from the Memory Bank Array. Through the write-and-read process, the input matrix is written in row-major order to the MBA and read in column-major order from the MBA.

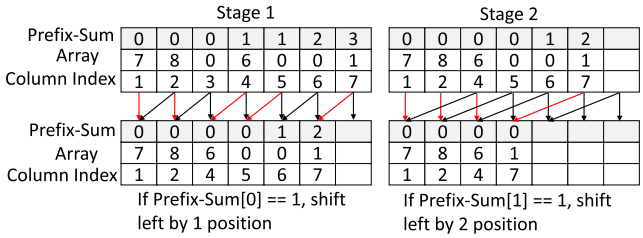


Fig. 8. Dense-to-sparse Module.

Data Format Management (DFM) Unit: DFM Unit transforms the data format between sparse format and dense format. It has two modules – Dense-to-sparse (D2S) Module and Sparse-to-dense (S2D) Module. Suppose the D2S Module can read n elements per clock cycle. Then, the D2S Module has $\log(n)$ pipeline stages. For an n-element array, we use the value of Prefix-Sum to indicate the number of zeros before an element in this array. An example is shown in Fig. 8. In Stage $i\,(1 \leqslant i \leqslant \log(n))$, an array element will be shifted left by 2^{i-1} positions if the $(i-1)$th bit of Prefix Sum value is equal to 1. The throughput of D2S Module is n elements per cycle. For example, a DDR4 channel of the FPGA board can output 16 32-bit data per cycle. A D2S Module of $n = 16$ is sufficient to match the data rate of a DDR4 channel. The architecture of S2D is similar to D2S, but in the reverse direction.

1) Analysis of Overhead: Data reorganization, including both data layout and format transformations, is performed during data loading from DDR memory, hiding the reorganization cost within the data loading process. Specifically, the Data Layout Management Unit (DLM) reads and outputs n data per cycle, while the Dense-to-sparse or Sparse-to-dense modules also process n data per cycle. By selecting n to match the DDR memory channel data rate (for example, a DDR4 memory channel outputs 16 32-bit data per clock cycle, so we set $n = 16$), data reorganization is executed in a streaming fashion as data is loaded. Therefore, the overhead of data reorganization is hidden by the data loading.

E. Instruction Set

We develop a customized instruction set, including computation instructions, memory read/write instructions. (1) *Computation Instructions* includes the instruction for each computation primitives (e.g., DDMM instruction). Each instruction contains the meta data (e.g., matrix size) of the corresponding computation primitive. The Instruction Decoder decodes the instruction and generates a control signal for the PE to execute the hardware primitives in pipelined manner. *Memory Read/Write Instructions* launch the data transactions between the on-chip buffer and the external memory.

VI. RUNTIME SYSTEM

The compiler generates a sequence of bytecode from the configuration of the input model and input data. At runtime, the runtime system loads the bytecode and performs just-in-time (JIT) compilation to generate instructions for hardware execution. During this JIT process, the runtime system performs

TABLE VI
PERFORMANCE MODEL OF EXECUTING A MATRIX MULTIPLICATION OPERATION $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$ USING THREE EXECUTION MODES

Execution Mode	MACs per cycle	Execution time (clock cycles)
GEMM	p_{aa}^2	$t_{\text{gemm}} = \frac{d_1 \times d_2 \times d_3}{p_{aa}^2}$
SpDMM	$p_{aa}^2/2$	$t_{\text{spdmm}} = \beta_{\text{min}} \times \frac{2 \times d_1 \times d_2 \times d_3}{p_{aa}^2}$, where $\beta_{\text{min}} = \text{Min}\,(\beta\,(\mathbf{X}), \beta\,(\mathbf{Y}))$
SPMM	p_{aa}	$t_{\text{spmm}} = \frac{\beta(\mathbf{X}) \times \beta(\mathbf{Y}) \times d_1 \times d_2 \times d_3}{p_{aa}}$

In our design, we set $paa > 4$.

two optimizations (1) *dynamic sparsity exploitation:* It exploits the data sparsity in matrix multiplication operation to reduce the total computation complexity, thus reducing the inference latency; (2) *dynamic task scheduling:* it automatically balances the workload between multiple processing elements to improve the runtime resource utilization.

A. Dynamic Sparsity Exploitation

In the matrix multiplication (MM) operations $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$ defined in the bytecode (See Table III), there can be data sparsity in the two input matrices $\mathbf{X}$ and $\mathbf{Y}$. Based on the sparsity of $\mathbf{X}$ and $\mathbf{Y}$, the runtime system can dynamically map the MM operation to the computation primitives, including dense-dense matrix multiplication (DDMM), sparse-dense matrix multiplication (SpDMM), and sparse-sparse matrix multiplication (SPMM). By exploiting the data sparsity, we can skip the zero elements and reduce the total computation complexity. Several design considerations enable our proposed dynamic sparsity exploitation strategy: (1) As the data sparsity of the intermediate results is unknown at compile time, the compiler generates the bytecode that does not depend on the data sparsity. (2) Our proposed accelerator incorporates the hardware sparsity profiler that can quickly profile the data sparsity of the intermediate results. (3) Our proposed accelerator design has a flexible data path and memory organization to support the three primitives (DDMM, SpDMM, and SPMM) that can execute matrix multiplication of various data sparsity.

For dynamic sparsity exploitation, we develop the performance model to decide how to map the matrix multiplication operation ($\mathbf{Z} = \mathbf{X} \times \mathbf{Y}, \mathbf{X} \in \mathbb{R}^{d_1 \times d_2}, \mathbf{Y} \in \mathbb{R}^{d_2 \times d_3}, \mathbf{X} \in \mathbb{R}^{d_1 \times d_3}$) to the computation primitives (DDMM, SpDMM, and SPMM). We define the data sparsity of a matrix $\mathbf{X}$ as $\beta(\mathbf{X})$ which is calculated through:

$$\beta(\mathbf{X}) = \frac{\text{\# of non-zero elements in } \mathbf{X}}{\text{total \# of elements in } \mathbf{X}} \tag{4}$$

The performance model for three execution modes (GEMM mode, SpDMM mode, and SPMM mode) is demonstrated in Table VI. To execute the matrix multiplication ($\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$, $\mathbf{X} \in \mathbb{R}^{d_1 \times d_2}$, $\mathbf{Y} \in \mathbb{R}^{d_2 \times d_3}$, $\mathbf{X} \in \mathbb{R}^{d_1 \times d_3}$), GEMM mode can execute p_{aa}^2 MAC operations per cycle (See Section VI-C). Therefore, GEMM mode takes $\frac{d_1 \times d_2 \times d_3}{p_{aa}^2}$ clock cycles to execute $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$. SpDMM mode can execute $p_{aa}^2/2$ MACs per clock cycle and can skip the zero elements in one input matrix (either $\mathbf{X}$ or $\mathbf{Y}$). Therefore, SpDMM mode takes $\beta_{\text{min}} \times \frac{2 \times d_1 \times d_2 \times d_3}{p_{aa}^2}$ clock cycles to execute $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$, where $\beta_{\text{min}} = \text{Min}(\beta(\mathbf{X}), \beta(\mathbf{Y}))$. SPMM mode can execute p_{aa} MACs per clock cycle and skip the zero elements in both input matrices. Therefore, SPMM mode takes at most $\frac{\beta(\mathbf{X}) \times \beta(\mathbf{Y}) \times d_1 \times d_2 \times d_3}{p_{aa}}$ clock cycles to execute $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$. Based on the performance model, we propose the dynamic mapping algorithm (Algorithm 6) that maps matrix multiplication operation to computation primitive based on data sparsity.

Algorithm 6: Dynamic Sparsity Exploitation Strategy.

Input: Input matrix $\mathbf{X}$ and input matrix $\mathbf{Y}$;
Output: Computation primitive (TargetPrimitive($\mathbf{X}$, $\mathbf{Y}$)) to execute $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$;

```
1: TargetPrimitive(X, Y) ← NULL
2: The buffers to store X and Y: buf_X, buf_Y
3: β_min = Min(β(X), β(Y))
4: β_max = Max(β(X), β(Y))
5: if β_min = 0 then                          ▷ Skip empty input matrix
6:   Skip the multiplication of X and Y
7: if β_min ⩾ 1/2 then                        ▷ Case 1
8:   TargetPrimitive(X, Y) ← GEMM
9:   buf_X ← VB1 and buf_Y ← VB2
10: else
11:   if β_max ⩾ 2/p_aa then                  ▷ Case 2
12:     TargetPrimitive(X, Y) ← SpDMM
13:     buf_argmin(β(M)) ← SB, (M ∈ {X, Y})
14:     buf_argmax(β(M)) ← VB1, (M ∈ {X, Y})
15:   else                                    ▷ Case 3
16:     TargetPrimitive(X, Y) ← SPMM
17:     buf_X ← SB and buf_Y ← VB1
```

Note that the two thresholds (in line 7 ($\beta_{\text{min}} \geqslant \frac{1}{2}$), and in line 11 ($\beta_{\text{max}} \geqslant \frac{2}{p_{aa}}$)) in Algorithm 6 are determined based on the performance model of t_{gemm}, t_{spdmm}, and t_{spmm} (Table VI). This leads to three cases in Algorithm 6: (1) Case 1: $\beta_{\text{min}} \geqslant \frac{1}{2}$, (2) Case 2: $\beta_{\text{min}} < \frac{1}{2}$ and $\beta_{\text{max}} \geqslant \frac{2}{p_{\text{aa}}}$, (3) Case 3: $\beta_{\text{min}} < \frac{1}{2}$ and $\beta_{\text{max}} < \frac{2}{p_{\text{aa}}}$. The above three cases are non-overlapping and cover the entire domain of β_{min} and β_{max}: $0 \leqslant \beta_{\text{min}} \leqslant \beta_{\text{max}} \leqslant 1$. For instance, when $\beta_{\text{min}} \geqslant \frac{1}{2}$ (Case 1), we can obtain that $t_{\text{gemm}} \leqslant t_{\text{spdmm}}$ and $t_{\text{gemm}} \leqslant t_{\text{spmm}}$. Therefore, GEMM mode leads to the smallest execution time for Case 1. Similarly, we derive that SpDMM mode and SPMM mode lead to the lowest execution time for Case 2 and Case 3, respectively.

B. Dynamic Task Scheduling

The runtime system is executed on the soft processor. At runtime, it takes the bytecode sequence as input and performs dynamic task scheduling for workload balance among the parallel PEs, as demonstrated in Algorithm 7. The runtime system schedules the workload for each matrix operation $\mathcal{M}(\mathbf{Z} = q(\mathbf{X}, \mathbf{Y}))$ in TQ one-by-one. For all the computation tasks $\{T(\mathbf{Z}_{ij}) : 1 \leqslant i \leqslant \frac{d_1}{N}, 1 \leqslant j \leqslant \frac{d_2}{N}\}$ in $\mathcal{M}(\mathbf{Z} = q(\mathbf{X}, \mathbf{Y}))$, the runtime system exploits a centralized load balancing scheme [50] to allocate the

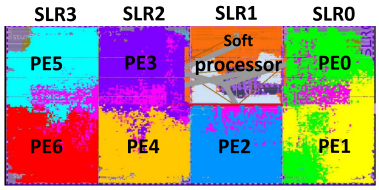


Fig. 9. Device map on Xilinx Alveo U250 FPGA board.

TABLE VII
HARDWARE RESOURCE UTILIZATION

	LUTs	DSPs	BRAMs	URAMs
Soft Processor	5.5K	6	26	0
One PE	118K	1296	192	120
FPGA shell	182K	13	447	0
Total	1008K	9091	1819	840
Available	1728K	12288	2688	960
Utilization	58%	73%	67.6%	87.5%

computation tasks to the parallel PEs. For each small matrix operation $\mathbb{M}$ in $T(\mathbf{Z}_{ij})$, the runtime system performs dynamic sparsity exploitation that maps this matrix operation to computation primitive based on the data sparsity. Based on the selected computation primitive, the runtime system generates the instruction sequence for hardware execution.

VII. Implementation Details

Implementation of hardware accelerator: We conduct comprehensive experiments to evaluate the performance of VisionAGILE. We implement the hardware design on a state-of-the-art FPGA platform, Xilinx Alveo U250, consisting of four Super Logic Regions (SLRs). The FPGA off-chip memory has four DDR4 memory banks with total 77 GB/s bandwidth. We implement the accelerator using Verilog to estimate the hardware resource. We implement 7 processing elements (PEs) with each SLR having 2 PEs except SLR1. Because half of SLR1 is occupied by the soft processor, FPGA shell, and memory subsystem. The size of ALU Array in each PE is $p_{\text{aa}} \times p_{\text{aa}} = 16 \times 16$. The soft processor is implemented using Xilinx Microblaze [51] IP core. We perform FPGA synthesis and place-route using Xilinx Vivado v2022.2. The generated device map is shown in Fig. 9. The hardware resources are obtained from Xilinx Vivado v2022 and are reported in Table VII. For performance evaluation, we build a cycle-accurate simulator to get the cycle count. In our design, we achieve a post-place-and-route frequency of 600 MHz for the DSPs and a frequency of 300 MHz for other components, following the methodology in AMD DPU [18]. The achieved post-place-and-route frequencies are used for simulation. We simulate the performance of external DDR memory using Ramulator [52].

Impact of Resource sharing: As discussed in Section VI-C, through resource sharing, different computation primitives share the same set of computation units (ALU Array, Activation Units), on-chip buffers (Scalar/Vector/Result Buffers), and routing networks (B2B network and B2P network). The wires of different primitives and multiplexers for selecting data paths incur extra area costs. In each PE, these wires and multiplexers consume 37K LUTs (Table VII), taking 31% LUTs consumption of a PE (A PE consumes 118K LUTs). Through resource sharing, our design only costs extra 31% LUTs for supporting various computation primitives.

Algorithm 7: Dynamic Task Scheduling.

Input: Input bytecode sequence that defines the task dependency graph TQ of the input model with each matrix operation partitioned;

1: **for** each matrix operation $\mathcal{M}(\mathbf{Z} = q(\mathbf{X}, \mathbf{Y}))$ in TQ **do**
2: **for** each computation task $T(\mathbf{Z}_{ij})$ in $\mathcal{M}(\mathbf{Z} = q(\mathbf{X}, \mathbf{Y}))$ **do**
3: **if** there is an idle PE_p **then**
4: Assign the workload of $T(\mathbf{Z}_{ij})$ to PE_p
5: **for** each small matrix operation $\mathbb{M}$ in $T(\mathbf{Z}_{ij})$ **do**
6: Map $\mathbb{M}$ to primitive using Algorithm 6
7: Generate instructions for executing the computation primitive
8: Execute the computation primitive on PE_p
9: Wait until all $T(\mathbf{Z}_{ij})$ in $\mathcal{M}(\mathbf{Z} = q(\mathbf{X}, \mathbf{Y}))$ is finished

Implementation of compiler: We develop the compiler using Python. The compiler takes as the input the user-defined ML models which are developed using PyTorch [53], PyTorch Geometric [49], and Hugging Face [54]. The compiler then generates an intermediate representation from the user-defined ML models. Each compilation step is wrapped as a function to apply to the intermediate representation. Finally, the compiler generates the bytecode for the runtime system.

Implementation of Runtime system: The runtime system is executed on the soft processor and developed using C++.

VIII. Evaluation Results

A. Benchmark, Datasets, Baselines, and Metrics

1) Benchmark: We conduct experiments on various ML models in computer vision tasks as shown in Table IX including CNNs, GNNs, GNN-based CV tasks (CNN+GNN), and ViTs. Note that we carefully select the benchmarks from diverse computer vision tasks (Table IX), which have various model sizes (Table XI), and the input data of various data modality and dimensions (Table XII). Evaluating the selected benchmarks, we expect the VisionAGILE will work similarly on a broad range of computer vision tasks.

2) Datasets: The datasets for evaluating various models are elaborated in Table X. The evaluated datasets are obtained from the original papers of these models.

3) Baselines: We compare the performance of VisionAGILE with various baseline platforms. First, we compare VisionAGILE with CPU and GPU since they are flexible and can support all models. Additionally, we compare VisionAGILE with state-of-the-art DSAs in their respective domains. For example, we compare VisionAGILE with state-of-the-art CNN DSAs, such as OPU [20] and DPU [18], on `c1-c5`.

TABLE VIII
SPECIFICATIONS OF PLATFORMS

Platforms	General Purpose Processors		CNN DSAs		GNN accelerators		VisionAGILE
	CPU	GPU	AMD DPU [18]	OPU1024 [20]	BoostGCN [31]	GraphAGILE [22]	
Platform	AMD Ryzen 3990x	Nvidia RTX A5000	ZCU102	Xilinx XC7K325T	Stratix10 GX	Alveo U250	Alveo U250
Platform Technology	TSMC 7 nm	Samsung 8 nm	TSMC 16 nm	28 nm	Intel 14 nm	TSMC 16 nm	TSMC 16 nm
Peak Performance	3.7 TFLOPS	27.7 TFLOPS	1.15 TFLOPS	0.2 TFLOPS	0.64 TFLOPS	0.64 TFLOPS	1.08 TFLOPS
On-chip Memory	256 MB L3 cache	6 MB L2 cache	32.1 MB	2 MB	45 MB	45 MB	45 MB
Memory Bandwidth	107 GB/s	768 GB/s	19.2 GB/s	12.8 GB/s	77 GB/s	77 GB/s	77 GB/s

TABLE IX
EVALUATED MODELS IN THE EXPERIMENTS

Category	Work	Notation	Task
CNN	AlexNet [2]	c1	Image Classification
	ResNet-50 [1]	c2	Image Classification
	ResNet-101 [1]	c3	Image Classification
	VGG16 [55]	c4	Image Classification
	VGG19 [55]	c5	Image Classification
GNN	GCN [4]	g1	Node Classification
	GraphSAGE [1]	g2	Node Classification
	GAT [1]	g3	Node Classification
CNN + GNN	[11]	b1	Few-shot image classification
	[12]	b2	Multi-label image classification
	[47]	b3	Image segmentation
	[56]	b4	Skeleton-based action recognition
	[57]	b5	SAR automatic target classification
ViT	ViT-Base [7]	V1	Image Classification
	ViT-Large [7]	V2	Image Classification
	ViT-Huge [7]	V3	Image Classification
	Swin-tiny [7]	V4	Image Classification, segementation
	Deit-small [7]	V5	Image Classification
	Deit-base [7]	V6	Image Classification

TABLE X
DATASETS AND BASELINE PLATFORMS

Models	Datasets	Baseline Platforms
c1, c2 c3, c4, c5	ImageNet (IN) [58]	CPU, GPU OPU [20], DPU [18]
g1, g2, g3	Cora (CO) [4], PubMed (PU) [4] CiteSeer (CI) [4] Flickr (FL) [5] Reddit (RE) [5], Yelp (YE) [59] AmazonProducts (AP) [59])	CPU, GPU BoostGCN [31] GraphAGILE [22]
b1	Omniglot (OM) [60]	CPU, GPU
b2	MS-COCO (MC) [61]	
b3	Cityscapes (City) [62]	
b4	NTU RGB+D (NR) [63]	
b5	MSTAR (MS) [64]	
v1, v2, v3 v4, v5, v6	ImageNet (IN) [58]	CPU, GPU

4) Metrics: We consider two performance metrics (1) *hardware execution latency:* this measures the accelerator's latency when batch size is 1. In applications like autonomous driving [65], the data captured by the sensors come frame-by-frame and each frame needs real-time processing for behavior planning. Low latency processing using batch size 1 is crucial for quick responses to changing conditions and ensuring safety. The measured latency is the time elapsed from when the input data, model weights, and bytecode are stored in the FPGA DDR memory until the inference results are written back to the FPGA DDR memory. The latency measurement excludes both the compilation time and the data (input model, input data, and bytecode) transfer time from the host CPU to the FPGA DDR memory. The same metric is used in measuring the performance of the baseline CPU and GPU platforms. (2) *throughput:* when comparing with state-of-the-art CNN accelerators for standalone CNNs, we use throughput as the performance metric. CNN accelerator performance is typically reported in terms of throughput [18], [20].

TABLE XI
SIZE OF THE MODELS

Model	Size (MB)	Model	Size (MB)	Model	Size (MB)
c1	244	g2-CO	0.184	g3-YE	0.026
c2	102	g2-PU	0.064	g3-AP	0.021
c3	178	g2-CI	0.474	b1	1.70
c4	553	g2-FL	0.064	b2	179.6
c5	574	g2-RE	0.082	b3	214
g1-CO	0.092	g2-YE	0.051	b4	12.39
g1-PU	0.032	g2-AP	0.039	b5	12.88
g1-CI	0.237	g3-CO	0.092	v1	346
g1-FL	0.032	g3-PU	0.032	v2	1216
g1-RE	0.041	g3-CI	0.237	v3	2528
g1-YE	0.026	g3-FL	0.032	v4	247
g1-AP	0.020	g3-RE	0.041	v5	264
				v6	346

TABLE XII
SIZE OF INPUT DATA IN VARIOUS DATASETS

Dataset	Modality	Dimension
ImageNet (IN)	Image	$224 \times 224 \times 3$ or $336 \times 336 \times 3$
Cora (CO)	Graph	V: 2708 E: 5429 F: 1433
PubMed (PU)	Graph	V: 19717 E: 44338 F: 500
Citeseer (CI)	Graph	V: 3327 E: 4732 F: 3703
Flickr (FL)	Graph	V: 89250 E: 899756 F: 500
Reddit (RE)	Graph	V: 232,965 E: 116,069,919 F: 602
Yelp (YE)	Graph	V: 716,847 E: 6,977,410 F: 300
Amazon-Products (AP)	Graph	V: 1,569,960 E: 264,339,468 F: 200
Omniglot (OM)	Image	105×105
MS-COCO (MC)	Image	640×488
Cityscapes (City)	Image	2048×1024
NTU RGB+D (NR)	Skeleton	$300 \times 25 \times 3$
MSTAR (MS)	Image	128×128

'V' means the number of vertices and 'E' means the number of edges.

B. Comparison With CPU and GPU Implementation

We compare the performance of VisionAGILE with state-of-the-art CPU and GPU implementations: (1) The CPU and GPU implementations of c1-c5 are from PyTorch library [53]; (2) The CPU and GPU implementations of g1-g3 are from PyTorch Geometric library [49]; (3) The CPU and GPU implementations of b1-b5 are from the open-soured implementation

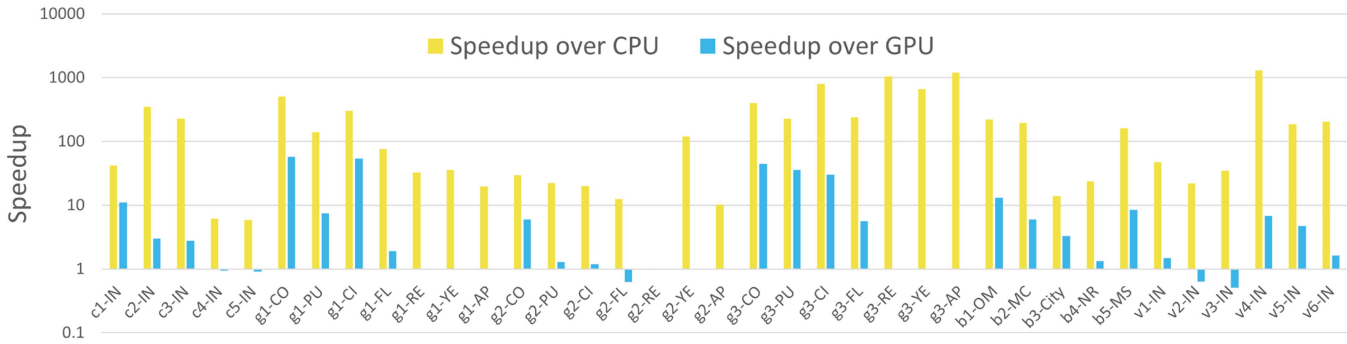


Fig. 10. Speedup over the state-of-the-art CPU and GPU implementations. Each bar shows the comparison on a specific model and datasets. For example, c1-IN means that execute c1 on ImageNet (IN) dataset. Note that in g1-RE, g1-YE, g1-AP, g2-RE, g2-YE, g2-AP, g3-RE, g3-YE and g3-AP, the GPU implementations run out of memory (OoM).We do not show these results in this figure.

TABLE XIII
AVERAGE SPEEDUP (BATCH SIZE 1 LATENCY) OVER CPU AND GPU

	c1-c3	g1-g3	b1-b5	v1-v6	Average
Speedup over CPU	41.2×	90.0×	74.7×	110.6×	81.7×
Speedup over GPU	2.4×	8.0×	4.9×	1.7×	4.8×

TABLE XIV
IMPACT OF DYNAMIC SPARSITY EXPLOITATION

Model + Dataset	Speedup	Model + Dataset	Speedup	Model + Dataset	Speedup
c1-IN	1.52×	g2-CO	1.13×	g3-YE	1.00×
c2-IN	1.04×	g2-PU	1.20×	g3-AP	1.00×
c3-IN	1.02×	g2-CI	1.08×	b1-OM	1.00×
c4-IN	1.57×	g2-FL	1.00×	b2-MC	1.01×
c5-IN	1.60×	g2-RE	1.00×	b3-City	1.27×
g1-CO	4.29×	g2-YE	1.00×	b4-NR	1.13×
g1-PU	1.85×	g2-AP	1.00×	b5-MS	1.00×
g1-CI	11.1×	g3-CO	19.1×	v1-IN	1.04×
g1-FL	1.02×	g3-PU	3.83×	v2-IN	1.07×
g1-RE	1.00×	g3-CI	3.91×	v3-IN	1.01×
g1-YE	1.00×	g3-FL	1.07×	v4-IN	1.01×
g1-AP	1.00×	g3-RE	1.00×	v5-IN	1.00×
Average	1.40×			v6-IN	1.03×

Each entry in the speedup column shows the achieved speedup of VisionAGILE with dynamic sparsity exploitation over VisionAGILE without dynamic sparsity exploitation.

of the original papers [66], [67], [68], [69], [70]; (4) The CPU and GPU implementations of v1-v6 are from Hugging Face [54]. The comparison results are demonstrated in Fig. 10, which shows the speed of inference when the batch size equals 1. On average, VisionAGILE achieves 81.7× and 4.8× speedup over state-of-the-art CPU and GPU implementations. The average speedup is summarized in Table XIII . The achieved speedup of VisionAGILE is due to several reasons: (1) VisionAGILE has a streaming architecture with fine-grained data parallelism. It leads to high computational efficiency when the model or input data size is small, such as c1-IN (see Tables XII and XI for data size and model size, respectively). In contrast, CPU and GPU utilize coarse-grained thread-level parallelism, which is more suitable for large models or input data, such as c5-IN and v3-IN. However, when it comes to the small model or input data, the CPU and GPU have large overhead for thread launch and synchronization overhead for enabling thread-level parallelism, which leads to high inference latency. (2) VisionAGILE has a customized on-chip memory organization, leading to low-latency on-chip memory access. On the contrary, the CPU and GPU have a complex cache hierarchy, leading to large on-chip memory access latency. Moreover, CPU or GPU have limited cache sizes (e.g., 32KB L1 cache and 512KB L2 cache). The data exchange among L1, L2, and L3 caches has significant overhead and results in reduced sustained performance. (3) VisionAGILE utilizes dynamic sparsity exploitation to reduce computational complexity at runtime, which further reduces inference latency. Dynamic sparsity exploitation requires collaboration between hardware design (e.g., data layout and data format management, sparsity profiler) and runtime system (e.g., performance model), which are hard to achieve on CPU and GPU.

Note that when the model or the input data is extremely large (For example, g2-FL, v2-IN, and v3-IN, See Fig. 10), GPU outperforms VisionAGILE because the large model and input data can enable massive thread-level parallelism for GPU execution. Moreover, GPU has much higher peak performance (27× higher) and external memory bandwidth (10× higher) than VisionAGILE. Therefore, GPU can efficiently deal with the massive computation parallelism of the large model.

1) Discussion on the Throughput of GPU: While VisionAGILE achieves lower latency when batch size is 1, GPU can achieve higher throughput by increasing the batch size (e.g., 8/16/32) as GPU has higher peak performance and memory bandwidth. The comparison results are shown in Fig. 11. GPU achieves 1.6×, 2.3×, and 2.8× higher throughput compared with VisionAGILE (with data sparsity exploitation) using batch size 4, 8, and 16, respectively. Nevertheless, this work targets latency-sensitive applications (e.g., autonomous driving). For higher throughput, it requires FPGA vendors to develop more powerful FPGA boards with more hardware resources.

C. Impact of Dynamic Sparsity Exploitation

1) Impact on the Performance: We evaluate the impact of dynamic sparsity exploitation by comparing the performance of VisionAGILE with dynamic sparsity exploitation (With-DSE) and

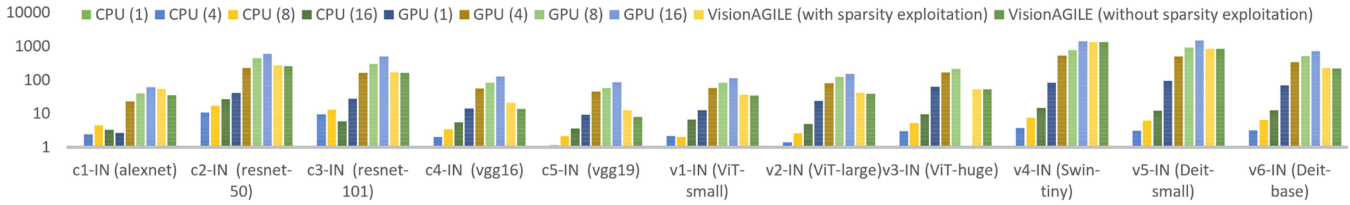


Fig. 11. Comparison of throughput (images/second) with baseline CPU and GPU. The y-axis is on a logarithmic scale. Note that CPU/GPU (x) indicates that the CPU/GPU executes the inference when the batch size is x. The throughput is divided by the throughput of GPU (1) for each dataset to show the normalized speedup. VisionAGILE uses the batch size of 1. Note the baseline GPU cannot execute the inference for ViT-huge when batch size is 16 due to the limited size of global memory.

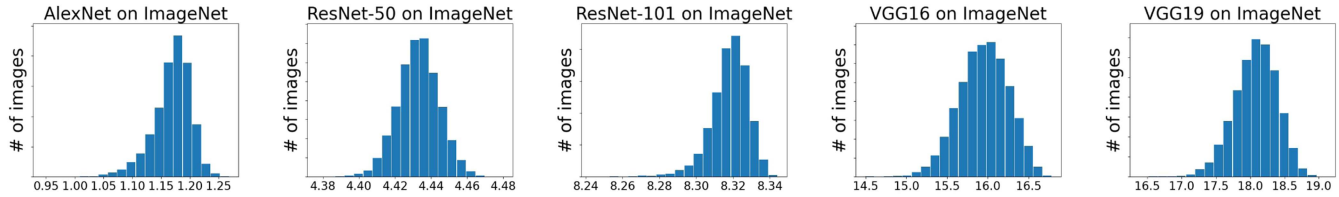


Fig. 12. Distribution of inference latency of `c1` (AlexNet), `c2` (ResNet50), `c3` (ResNet101), `c4` (VGG16), and `c5` (VGG19) on ImageNet dataset. X axis is the inference latency (ms).

TABLE XV
INFERENCE LATENCY (`c1`, `c2`, `c3`, `c4`, `c5`) WITHOUT DYNAMIC SPARSITY EXPLOITATION ON IMAGENET DATASET

Model	`c1`	`c2`	`c3`	`c4`	`c5`
Latency (ms)	1.79	4.62	8.34	25.2	29.2

TABLE XVI
COMPARISON OF THROUGHPUT (IMAGES/SECOND) WITH CNN ACCELERATORS ON VARIOUS CNN MODELS

	Throughput (unnormalized)					Computation efficiency of VisionAGILE over other accelerator
	`c1`	`c2`	`c3`	`c4`	`c5`	
DPU [18]	N/A	274	N/A	43.4	38.8	0.98×
OPU1024 [20]	N/A	54.4	27	12.2	9.7	0.90×
GCV-Turbo	512.9	254.7	127.3	58.8	46.5	1.15×
VisionAGILE	849.9	225.5	120.2	62.4	55.1	1×

TABLE XVII
COMPARISON OF HARDWARE EXECUTION LATENCY (MS) WITH STATE-OF-THE-ART GNN ACCELERATORS

	Latency (unnormalized)							Computation efficiency of VisionAGILE over the accelerator
	CO	**CI**	**PU**	**FL**	**RE**	**YE**	**AP**	
BoostGCN	0.0194	0.0253	0.1662	20.1	98.5	193.5	793.5	1.99×
GraphAGILE	0.819	2.55	2.24	11.5	97.2	104.3	315	18.4×
GCV-Turbo	0.48	1.47	1.25	6.09	72.7	43.5	196.9	17.8×
VsionAGILE	0.017	0.018	0.137	1.81	87.39	49.6	214.5	1×

VisionAGILE without dynamic sparsity exploitation (Without-DSE). In the case of VisionAGILE without dynamic sparsity exploitation, the runtime system directly maps the matrix multiplication operation to the hardware primitives (DDMM, SpDMM, SPMM) without taking into account data sparsity. Specifically, the runtime system assigns the matrix multiplication operations of the message passing layer to SpDMM, while mapping the matrix multiplication operations of other layers to DDMM.

The evaluation results are shown in Table XIV. On average, dynamic sparsity exploitation results in an average 1.44× speedup across various models and datasets. Note that the evaluated models are unpruned, indicating no data sparsity in weight matrices. The effectiveness of dynamic sparsity exploitation is particularly pronounced on specific datasets, such as CO, PU, and CI, which exhibit substantial data sparsity in their feature matrices. On some datasets and models (e.g., `g1`-RE), where there is limited data sparsity in the input data or intermediate results, no speedup is observed. We expect VisionAGILE to achieve higher speedup on the pruned models which have large sparsity in the model weights.

2) Fluctuation of Inference Latency: Due to the dynamic sparsity exploitation, the inference latency is related to the data sparsity of the intermediate results. For example, for CNNs or ViTs, different input images have different intermediate results in the intermediate layers, therefore leading to fluctuation of inference latency. Fig. 12 shows the distribution of inference latency of five CNNs (`c1`, `c2`, `c3`, `c4`, `c5`) on ImageNet dataset. Table XV summarizes the inference latency without dynamic sparsity exploitation. The experimental results demonstrate that (1) the inference latency fluctuates with the input data, (2) While the inference latency fluctuates, dynamic sparsity exploitation does not degrade the inference latency as indicated in Table XV, where Table XV shows the inference latency without dynamic sparsity exploitation.

D. Comparison With State-of-the-Art Accelerators

We compare the performance of VisionAGILE with CNN accelerators [18], [20], GNN accelerators [22], [31], and ViT accelerators within their respective domains. Different accelerators are implemented on different hardware platforms and use different amount of hardware resources. For a fair comparison, we normalize the performance (latency or throughput) by their respective peak performance (FLOPs), following the state-of-the-art

TABLE XVIII
COMPARISON ON VARIOUS VIT MODELS (LATENCY / THROUGHPUT)

	Peak Performance	v1 (ViT-small)	v2 (ViT-large)	v3 (ViT-huge)	v4 (Swin-tiny)	v5 (Deit-small)	v6 (Deit-base)	
		(ms) / (fps)	(ms) / (fps)	(ms) / (fps)	(ms) / (fps)	(ms) / (fps)	(ms) / (fps)	Speedup Over CPU (Latency / Throughput)
CPU	3.7 TFLOPS	712 / 7.4	1496 / 3.3	5253 / 1.5	4344 / 18.1	7373 / 13.1	3114 / 4.8	1× / 1×
GPU (RTX A5000)	27.7 TFLOPS	22.0 / 162.7	42.8 / 69.6	76.3 / 16.3	22.6 / 356	18.6 / 517	24.7 / 205	82.3× / 26.1×
VisionAGILE	1.08 TFLOPS	14.9 / 67.1	67.6 / 14.8	150.6 / 6.6	5.4 / 199	5.7 / 173.1	21.7 / 45.9	299.3× / 11.2×

works [71], [72]. For example, normalized throughput is calculated by: Normalized Throughput of [X] $= \frac{\text{Throughput of [X]}}{\text{Peak performance of [X]}}$ where [X] can be AMD DPU [18], OPU [20], BoostGCN [31], and GraphAGILE [22]. We define the normalized throughput/latency as *computation efficiency*.

1) Comparison With CNN Accelerators: We compare the performance of VisionAGILE with CNN accelerators [18], [20] on the CNN models c1–c5. The reported performance of [18], [20] is in throughput. The throughput of VisionAGILE is calculated by $\frac{1}{\text{latency}}$. As shown in Table XVI, VisionAGILE achieves the computation efficiency of $0.98\times$, $0.9\times$, and $1.15\times$ compared to DPU, OPU and GCV-Turbo, respectively. The results indicate that VisionAGILE achieves comparable computation efficiency in various CNN models. Compared with DPU and OPU, VisionAGILE achieves slightly lower computation efficiency due to two design trade-offs: (1) VisionAGILE's versatility, which supports various models, including CNNs, GNNs, and ViTs, sacrificing some CNN-specific architectural optimizations. For example, OPU's multi-level parallelism is fine-tuned for CNN convolution operations, whereas VisionAGILE's architecture is more generalized. (2) VisionAGILE's compilation flow is generalized for various models but cannot support specific convolution-specific optimizations. DPU selects dataflow for convolutional layers based on kernel size, which cannot be directly applied to GNN layers.

Compared with GCV-Turbo, VisionAGILE achieves higher computation efficiency because VisionAGILE utilizes the dynamic sparsity exploitation strategy to reduce the computation complexity at runtime. When there is a large data sparsity in the intermediate results (e.g., c1, c4, and c5), VisionAGILE achieves significant performance improvement. Note that the evaluated CNN models are unpruned with no data sparsity in the weight matrices. We expect that VisionAGILE achieves higher speedup on the pruned CNN models.

2) Comparison With GNN Accelerators: We compare VisionAGILE with the state-of-the-art GNN accelerators, BoostGCN [31], GraphAGILE [22], and GCV-Turbo. Latency measurements follow the methodology from [22], [31], focusing on a two-layer GCN model and graph datasets shown in Table XII. Table XVII presents the results, where latency is normalized by the peak performance of the accelerator to obtain the speedup. VisionAGILE achieves $1.99\times$, $18.4\times$, and $17.8\times$ computation efficiency of BoostGCN, GraphAGILE, and GCV-Turbo. The superior performance compared with BoostGCN can be attributed to BoostGCN's separate hardware modules for sparse and dense computation in GNNs, leading to the underutilization of runtime resources. In contrast, VisionAGILE optimizes resource utilization by employing a unified architecture for sparse and dense computations in GNNs. The higher performance over GraphAGILE and GCV-Turbo is due to VisionAGILE's utilization of dynamic sparsity exploitation that exploits the data sparsity in the intermediate results and input feature matrices, leading to significant performance improvement on the graph dataset with high data sparsity.

3) Comparison With ViT Accelerators: We compare the performance of VisionAGILE with CPU and GPU on various ViT models v1–v5. The experimental results are demonstrated in Table XVIII. Compared with the baseline GPU platform, VisionAGILE achieves lower latency ($0.27\times$) but lower throughput ($0.43\times$). Because GPU has higher peak performance ($25.6\times$) and massive thread-level parallelism, GPU can perform multiple input images in parallel.

Note that the evaluated ViT models are unpruned, so there is limited data sparsity in weight matrices or intermediate results. Since VisionAGILE exploits the dynamic data sparsity, we expect VisionAGILE to achieve higher throughput on pruned ViT models than CPU and GPU. While AMD DPU [18] can also execute ViT, there is no reported performance in [73] for ViT on DPU. Therefore, we leave the comparison with AMD DPU as the future work.

IX. CONCLUSION AND FUTURE WORK

In this paper, we proposed VisionAGILE, a versatile domain-specific accelerator for computer vision tasks. VisionAGILE has a flexible hardware architecture design to support various machine learning models in computer vision, including CNNs, GNNs, and ViTs. VisionAGILE has a unified compilation workflow to map various ML models on the proposed hardware accelerator. The runtime system of VisionAGILE performs dynamic sparsity exploitation to reduce the complexity that reduces the inference latency. The experimental results showed that VisionAGILE achieved $81.7\times$ and $4.8\times$ speedup compared with state-of-the-art CPU and GPU implementation on various ML models in computer vision, respectively. Moreover, VisionAGILE achieved a comparable or higher performance than the state-of-the-art CNN, GNN, or ViT accelerators. In the future, we plan to extend the VisionAGILE to AMD versal ACAP platform, which integrates AI engines with high peak performance for matrix operations. By exploiting AI engines for matrix operations, we expect to achieve higher peak performance.

Distribution Statement A: Approved for public release. Distribution is unlimited.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[3] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 779–788.

[4] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. Int. Conf. Learn. Representations*, 2016. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[5] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1025–1035.

[6] P. Veličković et al., "Graph attention networks," in *Proc. Int. Conf. on Learn. Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[7] A. Dosovitskiy et al., "An image is worth 16x16 words: Transformers for image recognition at scale," in *Proc. Int. Conf. Learn. Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=YicbFdNTTy

[8] Z. Liu et al., "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2021, pp. 9992–10 002.

[9] B. Kayalibay, G. Jensen, and P. van der Smagt, "CNN-based segmentation of medical imaging data," 2017, *arXiv: 1701.03056*.

[10] C. R. Qi, H. Su, M. Kaichun, and L. J. Guibas, "PointNet: Deep learning on point sets for 3D classification and segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 652–660.

[11] V. Garcia and J. Bruna, "Few-shot learning with graph neural networks," in *Proc. 6th Int. Conf. Learn. Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=BJj6qGbRW

[12] Z.-M. Chen, X.-S. Wei, P. Wang, and Y. Guo, "Multi-label image recognition with graph convolutional networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 5177–5186.

[13] T. Xiao et al., "Early convolutions help transformers see better," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 30 392–30 400.

[14] S. Li, C. Wu, and N. Xiong, "Hybrid architecture based on CNN and transformer for strip steel surface defect classification," *Electronics*, vol. 11, no. 8, 2022, Art. no. 1200.

[15] Google cloud. 2011. [Online]. Available: https://cloud.google.com/bigquery/docs/inference-overview

[16] Gaws inferentia: High performance at the lowest cost in Amazon EC2 for deep learning inference. 2019. [Online]. Available: https://aws.amazon.com/machine-learning/inferentia/

[17] Habana. 2022. [Online]. Available: https://habana.ai/

[18] DPU. 2018. [Online]. Available: https://docs.amd.com/r/en-US/pg338-dpu/Reference-Clock-Generation

[19] M. S. Abdelfattah et al., "DLA: Compiler and FPGA overlay for neural network inference acceleration," in *Proc. IEEE 28th Int. Conf. Field Programmable Log. Appl.*, 2018, pp. 411–4117.

[20] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "OPU: An FPGA-based overlay processor for convolutional neural networks," *IEEE Trans. Very Large Scale Integration Syst.*, vol. 28, no. 1, pp. 35–47, 2019.

[21] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, May/Jun. 2018.

[22] B. Zhang, H. Zeng, and V. Prasanna, "GraphAGILE: An FPGA-based overlay accelerator for low-latency GNN inference," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 9, pp. 2580–2597, Sep. 2023.

[23] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Commun. ACM*, vol. 63, no. 7, pp. 48–57, 2020.

[24] N. Jouppi et al., "TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *Proc. 50th Annu. Int. Symp. Comput. Archit.*, New York, NY, USA, 2023, pp. 1–14.

[25] E. Qin et al., "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 58–70.

[26] Y. Chen et al., "DaDianNao: A machine-learning supercomputer," in *Proc. IEEE 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 609–622.

[27] T. Luo et al., "DaDianNao: A neural network supercomputer," *IEEE Trans. Comput.*, vol. 66, no. 1, pp. 73–88, Jan. 2017.

[28] B. Zhang, H. Zeng, and V. Prasanna, "Hardware acceleration of large scale GCN inference," in *Proc. IEEE 31st Int. Conf. Appl.-Specific Syst. Archit. Process.*, 2020, pp. 61–68.

[29] M. Yan et al., "HyGCN: A GCN accelerator with hybrid architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 15–29.

[30] T. Geng et al., "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2020, pp. 922–936.

[31] B. Zhang, R. Kannan, and V. Prasanna, "BoostGCN: A framework for optimizing GCN inference on FPGA," in *Proc. IEEE 29th Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2021, pp. 29–39.

[32] S. Liang, C. Liu, Y. Wang, H. Li, and X. Li, "DeepBurning-GL: An automated framework for generating graph neural network accelerators," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des.*, 2020, pp. 1–9.

[33] H. Zeng and V. Prasanna, "GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2020, pp. 255–265.

[34] Y.-C. Lin, B. Zhang, and V. Prasanna, "HP-GNN: Generating high throughput GNN training implementation on CPU-FPGA heterogeneous platform," 2021, *arXiv:2112.11684*.

[35] T. Geng et al., "I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2021, pp. 1051–1063.

[36] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *Proc. 57th ACM/IEEE Des. Automat. Conf.*, 2020, pp. 1–6.

[37] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao, "FlowGNN: A dataflow architecture for real-time workload-agnostic graph neural network inference," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2023, pp. 1099–1112.

[38] M. Sun et al., "VAQF: Fully automatic software-hardware co-design framework for low-bit vision transformer," 2022, *arXiv:2201.06618*.

[39] T. Wang et al., "ViA: A novel vision-transformer accelerator based on FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 11, pp. 4088–4099, Nov. 2022.

[40] H. You et al., "ViTCoD: Vision transformer acceleration via dedicated algorithm and accelerator co-design," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2023, pp. 273–286.

[41] S. Nag et al., "ViTA: A vision transformer inference accelerator for edge applications," 2023, *arXiv:2302.09108*.

[42] Z. Lit et al., "Auto-viT-Acc: An FPGA-aware automatic acceleration framework for vision transformer with mixed-scheme quantization," in *Proc. 32nd Int. Conf. Field-Programmable Log. Appl.*, 2022, pp. 109–116.

[43] H. Fujiyoshi, T. Hirakawa, and T. Yamashita, "Deep learning-based image recognition for autonomous driving," *IATSS Res.*, vol. 43, no. 4, pp. 244–252, 2019.

[44] X. Huang et al., "The apolloscape dataset for autonomous driving," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, 2018, pp. 954–960.

[45] C. Chen et al., "A survey on graph neural networks and graph transformers in computer vision: A task-oriented perspective," 2022, *arXiv:2209.13232*.

[46] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, "High-performance low-memory lowering: GEMM-based algorithms for DNN convolution," in *2020 IEEE 32nd Int. Symp. Comput. Archit. High Perform. Comput.*, 2020, pp. 99–106.

[47] L. Zhang et al., "Dual graph convolutional network for semantic segmentation," in *Proc. Brit. Mach. Vis. Conf.*, 2019. [Online]. Available: https://ora.ox.ac.uk/objects/uuid:405e987d-9ff1-44ee-a4ac-290b35ed36a2

[48] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Proc. 10th Int. Workshop Front. Handwriting Recognit.*, 2006. [Online]. Available: https://inria.hal.science/inria-00112631/

[49] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *Proc. ICLR Workshop Representation Learn. Graphs Manifolds*, 2019. [Online]. Available: https://scholar.google.com/scholar?cluster=8974784740086755370&hl=en&as_sdt=0,5

[50] V. Kumar et al., *Introduction to Parallel Computing*, vol. 110. Redwood City, CA, USA: Benjamin/Cummings, 1994.

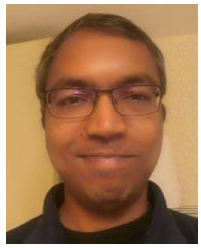
[51] Microblaze. 2021. [Online]. Available: https://docs.xilinx.com/v/u/2021.1-English/ug984-vivado-microblaze-ref

[52] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 45–49, Jan./Jun. 2016.

[53] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8026–8037.

[54] T. Wolf et al., "Huggingface's transformers: State-of-the-art natural language processing," 2019, *arXiv: 1910.03771*.
[55] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015. [Online]. Available: https://scholar.google.com/scholar?hl=en&as_sdt=0%2C5&q=Very+deep+convolutional+networks+for+1240+large-scale+image+recognition&btnG=
[56] S. Yan, Y. Xiong, and D. Lin, "Spatial temporal graph convolutional networks for skeleton-based action recognition," in *Proc. AAAI Conf. Artif. Intell.*, 2018, pp. 7444–7452.
[57] B. Zhang et al., "Graph neural network based SAR automatic target recognition with human-in-the-loop," in *Algorithms for Synthetic Aperture Radar Imagery XXX*, vol. 12520. Bellingham, WA, USA: SPIE, 2023, pp. 196–198.
[58] J. Deng, W. Dong, R. Socher, L. -J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.
[59] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graph-SAINT: Graph sampling based inductive learning method," in *Proc. Int. Conf. Learn. Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=BJe8pkHFwS
[60] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "The omniglot challenge: A 3-year progress report," *Curr. Opin. Behav. Sci.*, vol. 29, pp. 97–104, 2019.
[61] T.-Y. Lin et al., "Microsoft COCO: Common objects in context," in *Proc. 13th Eur. Conf. Comput. Vis.*, Zurich, Switzerland, Springer, 2014, pp. 740–755.
[62] M. Cordts et al., "The cityscapes dataset for semantic urban scene understanding," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 3213–3223.
[63] J. Liu, A. Shahroudy, M. Perez, G. Wang, L. -Y. Duan, and A. C. Kot, "NTU RGB+D 120: A large-scale benchmark for 3D human activity understanding," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 10, pp. 2684–2701, Oct. 2020.
[64] MStar. 1995. [Online]. Available: https://www.sdms.afrl.af.mil/index.php?collection=mstar
[65] K. Roszyk, M. R. Nowicki, and P. Skrzypczyński, "Adopting the YOLOv4 architecture for low-latency multispectral pedestrian detection in autonomous driving," *Sensors*, vol. 22, no. 3, 2022, Art. no. 1082.
[66] CPU and GPU implementation of few-shot image classification. 2017. [Online]. Available: https://github.com/vgsatorras/few-shot-gnn
[67] CPU and GPU implementation of multi-label image classification. 2019. [Online]. Available: https://github.com/megvii-research/ML-GCN
[68] CPU and GPU implementation of image segmentation. 2019. [Online]. Available: https://github.com/lxtGH/GALD-DGCNet
[69] CPU and GPU implementation of skeleton-based action recognition. 2018. [Online]. Available: https://github.com/yysijie/st-gcn
[70] CPU and GPU implementation of point cloud classification. 2020. [Online]. Available: https://github.com/WeijingShi/Point-GNN
[71] X. Chen et al., "ThunderGP: HLS-based graph processing framework on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2021, pp. 69–80.
[72] S. Zeng et al., "FlightLLM: Efficient large language model inference with a complete mapping flow on FPGA," 2024, *arXiv:2401.03868*.
[73] AMD DPU performance. 2023. [Online]. Available: https://xilinx.github.io/Vitis-AI/3.5/html/docs/reference/ModelZoo_Github_web.htm

Bingyi Zhang received the BS degree in microelectronics from Fudan University in 2017 and the MS degree in integrated circuit engineering from Fudan University in 2019 and the PhD degree in computer engineering from the University of Southern California (USC) in 2024. He is currently working as a developer technology engineer - AI with Nvidia. His research interests include high performance computing, parallel programming, and computer architecture.



Rajgopal Kannan received the BTech degree in computer science and engineering from IITBombay, in 1991 and the PhD degree in computer science from the University of Denver, in 1996. He is currently the branch chief and external competency co lead of Military Information Sciences with the Army Research Office (ARO) and a research adjunct professor in electrical engineering with the University of Southern California. He was formerly a professor with the Department of Computer Science, Louisiana State University (2000-2015). His academic research was funded by DARPA, NSF and DOE and he has published more than 150 research papers in international journals and conferences with two patents awarded in the area of network optimization. His research interests are at the intersection of graph analytics, machine learning and edge computing - enabling application acceleration at the edge on low power devices, for example using Software-Defined Memory for memory bound applications. He is also interested in cyber-physical systems, especially data-driven models and analytics driving Smartgrid optimization and control.

Carl Busart received the BS and MS degrees from Johns Hopkins University, the MBA degree from the University of Maryland, College Park, and the DEng degree from George Washington University. He is a branch chief with the U.S. Army Research Laboratory and a member of the Institute of Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM). His research interests include artificial intelligence / machine learning (AI/ML) and secure design.

Viktor K. Prasanna (Life Fellow, IEEE) received the BS degree in electronics engineering from Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the PhD degree in computer science from the Pennsylvania State University. He is Charles Lee Powell chair in engineering with the Ming Hsieh Department of Electrical and Computer Engineering and Professor of computer science at the University of Southern California (USC). His research interests include high performance computing, parallel and distributed systems, reconfigurable computing, and embedded systems. He serves as the director of the Center for Energy Informatics with USC. He served as the editor-in-chief of the IEEE TRANSACTIONS ON COMPUTERS during 2003–06. Currently, he is the editor-in-chief of the *Journal of Parallel and Distributed Computing*. He was the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the steering chair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and is the steering chair of the IEEE International Conference on High Performance Computing (HiPC). He received the 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University, the 2019 Distinguished Alumnus Award from the Indian Institute of Science (IISc) and the 2016 Distinguished Alumnus Award from the University Visvesvaraya College of Engineering (UVCE), Bangalore University. He received the W. Wallace McDowell Award from the IEEE Computer Society, in 2015 for his contributions to reconfigurable computing. He is a fellow of the ACM, and the American Association for Advancement of Science (AAAS). He is an elected member of Academia Europaea.