Implementing Neural Networks on Nonvolatile FPGAs With Reprogramming

Hao Zhang, *Student Member, IEEE*, Jian Zuo, Huichuan Zheng, Sijia Liu, Meihan Luo, and Mengying Zhao

***Abstract*—NV-FPGAs have attracted significant attention in research due to their high density, low leakage power, and reduced error rates. The nonvolatile memory (NVM) crossbar's compute-in-memory (CiM) capability further enables NV-FPGAs to execute high-efficiency, high-throughput neural network (NN) inference tasks. However, with the rapid increase in network size and considering that the parameter size often exceeds the memory capacity of the field programmable gate array (FPGA), implementing the entire network on a single FPGA chip becomes impractical. In this article, we utilize FPGA's inherent run time reprogramming feature to implement oversized NNs on NV-FPGAs. This approach splits NN models into multiple tasks for the cyclical execution. Specifically, we propose a performance-driven task adapter (PD-Adapter), which aims to achieve high-performance NN inference by employing the task deployment to optimize settings, such as processing element size and quantity, and the task switching to select the most suitable switching type for each task. We integrate the proposed PD-Adapter into an open-source toolchain and evaluate it. Experimental results demonstrate that the PD-Adapter can achieve a run time reduction of 85.37% and 76.12% compared to the baseline and execution-time-first policy, respectively.**

***Index Terms*—Compute-in-memory (CiM), neural network (NN) inference, nonvolatile field programmable gate arrays (FPGAs), nonvolatile memory.**

I. Introduction

Field programmable gate arrays (FPGAs) have become prominent in big data, edge computing, and image processing due to their flexibility and energy efficiency. Their inherent programmability enables seamless adaptation to these applications' diverse and evolving requirements, providing a cost-effective and highly adaptable solution. Significant advances in chip process technology over the past two decades have resulted in a downscaling from the micrometer to nanometer scale, facilitating the integration of increased computational and memory resources to meet the demands of large-data applications. However, this progression presents considerable challenges for the traditional static random-access memory (SRAM)-based FPGAs, primarily due to the SRAM's limited density and substantial leakage power. In recent years, emerging nonvolatile memories (NVMs) have offered new avenues for the FPGA design enhancement. Compared to SRAMs, NVMs provide greater scalability, lower leakage power, nonvolatility, and superior error resistance. The feasibility of integrating various NVMs, such as phase change memory (PCM) [1], resistive RAM (RRAM) [2], and magnetic RAM (MRAM) [3], into FPGA has been successfully demonstrated [1], [4], [5], [6], [7], [8], [9].

Manuscript received 10 August 2024; accepted 10 August 2024. This work was supported in part by the Natural Science Foundation of China (NSFC) under Grant 62372270. This article was presented at the International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Mengying Zhao.)*

The authors are with the School of Computer Science and Technology, Shandong University, Qingdao 266237, China (e-mail: zhaomengying@sdu.edu.cn).

Digital Object Identifier 10.1109/TCAD.2024.3443708

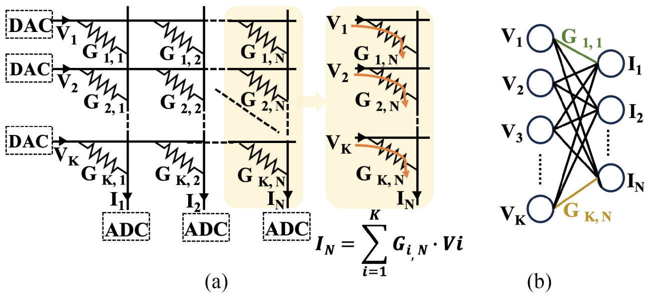

Fig. 1. Typical NVM-based CiM architecture. (a) Schematic of a crossbar architecture. (b) CiM for neural computation.

Furthermore, the NVM crossbar architecture intrinsically supports in-memory computing [10], [11], [12], [13]. Fig. 1 illustrates the ability of the NVM crossbar to execute the matrix-vector multiplication (MVM) operations with significant parallelism by mapping weights to the conductance of NVM cells and synchronizing vectors with the input voltage. Several research efforts have exploited this characteristic to equip NV-FPGAs with compute-in-memory (CiM) capability. For example, Ji et al. [14] integrated NVM crossbars into FPGA chips. Zha and Li [15] developed an NVM-based multifunctional block by transforming configurable blocks and routing structures. Zhang et al. [16] modified the structure of the on-chip memory, enhancing its computational functionality. These NVM crossbars and peripheral circuits integrated into the FPGA are referred to as CiM blocks. The combination of the CiM block's high parallel operator execution capability and the FPGA's flexible function configuration ability allows NV-FPGAs to meet the high-throughput computational demands of the neural network (NN) inference. Fig. 2 illustrates how these studies establish one or more CiM blocks as processing elements (PEs) units essential for NN computation on the NV-FPGA chip. The weight parameters are preprogrammed into the CiM blocks, and during operation, only the input vector representing the feature map is transmitted to the PE units based on the

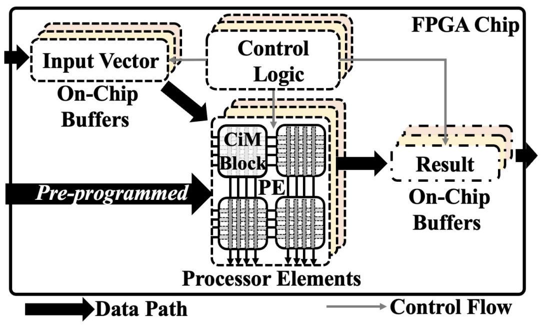

Fig. 2. CiM-block-based MVM PE implementation in NV-FPGA.

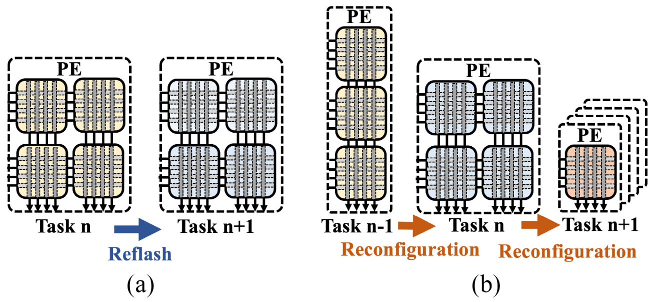

Fig. 3. Task switching by reflash or reconfiguration. (a) Reprogramming only the weight parameters used in task n+1. (b) Reprogramming both the weight parameters and connection.

CiM blocks, enabling high-throughput MVM. The CiM block enhances NV-FPGAs' ability to handle the large-scale MVMs, leading to higher throughput in the NN inference. However, these studies typically assume that the CiM block resources in NV-FPGAs are extensive enough to accommodate all the NN parameters. As NNs evolve and increase in scale, their parameters also increase correspondingly. This enhances their ability to learn the complex patterns and perform a broader range of tasks, but often exceeds the available resources of CiM blocks in NV-FPGAs.

The inherent run time reprogramming feature of FPGAs has been widely applied in run time configurable systems and run time context switching systems [17], [18], [19], [20]. In this article, we employ this feature to introduce an NN inference implementation designed to deploy oversized NNs on the resource-limited platforms. We segment the NN model into multiple tasks, with each task encompassing the implementation of PE for the corresponding layer. These tasks are sequentially reprogrammed into the FPGA chip to enable the task switching during run time. Task switching is divided into two categories based on the content of reprogramming: 1) "reflash" and 2) "reconfiguration." As shown in Fig. 3(a), reflash refers to task switching that utilizes the PE structure of the previous task, only reprogramming the weight parameters in the CiM blocks. However, when the PE size of the previous task does not meet the computational requirements of the current task or is inefficient, task switching needs to be achieved through reconfiguration. As shown in Fig. 3(b), reconfiguration refers to the task switching that involves constructing new PE implementations, entailing changes in the weight parameters, the size of the PEs, and the number of PEs. The amount of data that needs to be reprogrammed is different for these two task switching types, and the time cost is also different. We can flexibly design the choice between reflash and reconfiguration, aiming to reduce the overall run time while implementing the oversized NNs. The main contributions are as follows.

1) We utilize the inherent run time reprogramming feature of FPGAs to introduce an NN inference implementation tailored for the oversized NNs on the NV-FPGAs.
2) We propose the performance-driven task adapter (the PD-Adapter), aiming to optimize the NN inference performance through the strategic task deployment and task switching, thereby enhancing both the task execution efficiency and the task switching time.
3) We integrate the proposed PD-Adapter into an open-source FPGA synthesis toolchain and evaluate its effectiveness on the CiM block-equipped NV-FPGA platform.

The remainder of this article is organized as follows. Section II reviews the preliminaries of NV-FPGA and summarizes the related work. Section III shows motivation examples to briefly explain the task deployment and switching. Section IV details the implementation of the oversized NNs on the NV-FPGAs. Section V presents the evaluation results, followed by the conclusion in Section VI.

II. Preliminaries and Related Work

In this section, we first introduce the background of the FPGA architecture and programming techniques. Then, we summarize the related works involving CiM and run time switching on the NV-FPGAs.

A. FPGA Architecture and Programming Techniques

Fig. 4 illustrates a conventional island-style FPGA architecture, which primarily consists of the configurable elements, including configurable logic blocks (CLBs), connection boxes (CBs), and switch boxes (SBs). Each of these components contains a series of memory cells, and the required logical functions are achieved by preprogramming the memory cells. To accommodate escalating computational and buffering requirements, contemporary FPGAs incorporate heterogeneous resources like the block random access memories (BRAMs) and digital signal processors (DSPs) directly onto the FPGA die. CLBs can implement both the combinational and sequential logics. SBs and CBs, strategically positioned throughout the FPGA chip, facilitate versatile connectivity among the computational units, memory resources, and I/O interfaces. Due to the advantages of nonvolatility, high density, and near-zero leakage power, emerging NVMs are proposed to replace the current SRAMs in FPGA platforms, leading to designs of nonvolatile FPGAs [3], [21], [22]. Furthermore, the NVM-based CiM blocks are introduced to enhance the FPGA's processing capability, utilizing the electrical characteristics of the NVM crossbars to achieve highly parallel, low-power in-situ computation operations [14], [15], [16]. Like the traditional resources, such as DSPs and CLBs, the CiM blocks are distributed throughout the FPGA.

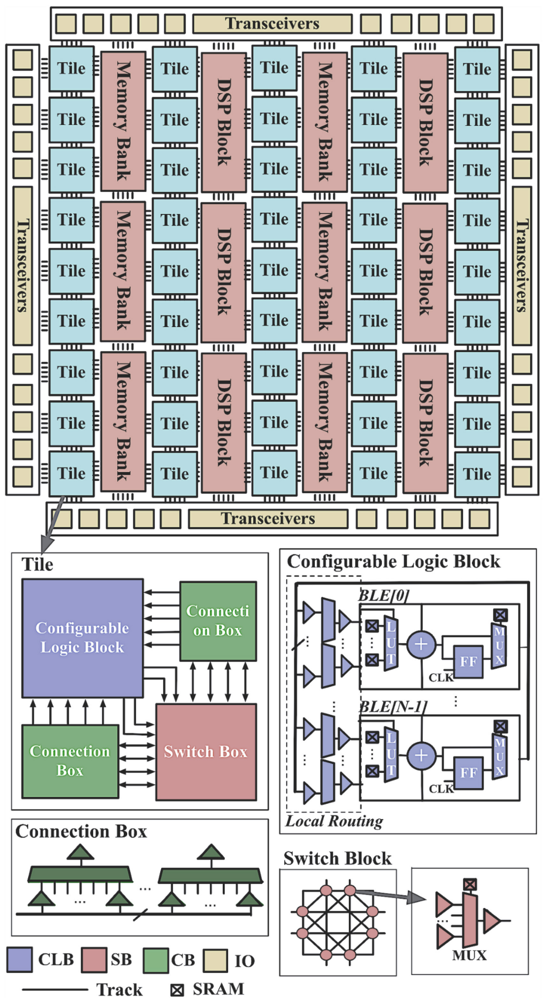

Fig. 4. Typical island-style FPGA architecture [23].

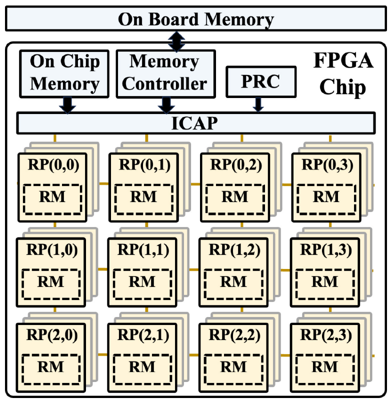

Fig. 5. Partial reconfiguration on FPGA. By utilizing the ICAP and partial PRC, configuration data are fetched from memory, enabling the dynamic alteration of the functionality of associated reconfiguration modules (RMs).

The applications will be implemented on the FPGA by the logic and physical synthesis. In the logic synthesis stage, high-level logic functions are converted into basic logic elements that can be implemented with the physical blocks. This stage involves optimization to minimize the number of logic elements and enhance the circuit efficiency. The physical synthesis stage maps the logic elements onto the FPGA's physical resources by placing these elements into the appropriate physical blocks and generating the routing to connect them. The final output of the synthesis process is the bitstream files that record the configuration data for the FPGA platform.

The traditional model of the FPGA usage involved configuring the device once, typically during the system startup, after which the FPGA would perform its designated function without change. However, as computing demands grew, especially in fields requiring adaptability and real-time processing, the concept of run time reprogramming emerged. Run time reprogramming allows an FPGA to be reconfigured while it is still operational, enabling dynamic adaptation to different tasks or algorithms without the need to power down or restart the system. Additionally, modern FPGAs support partial reprogramming as illustrated in Fig. 5, which allows for updating partial areas on the chip. The reprogramming time is proportional to the size of the reprogramming data. Both reflash and reconfiguration can be achieved through the partial programming. The former only requires programming the data within the PE, while the latter involves the programming of both the PE's data and related CLBs and SBs.

B. NV-FPGA With Compute-in-Memory

In nonvolatile FPGAs, SRAM-based memory cells are replaced with NVM-based memory cells. Several researchers have focused on the architecture design of nonvolatile FPGAs, such as PCM-based FPGA, STT-RAM-based FPGA, and RRAM-based FPGA. Architectures are proposed for the nonvolatile CLBs [24], nonvolatile SBs [25], and nonvolatile BRAMs [7]. Furthermore, some works have leveraged the in-memory computing characteristics of NVM by introducing NVM-based CiM function blocks into the FPGA chips. On the one hand, some research efforts are enhancing the existing CLBs or BRAMs to equip them with the CiM capabilities. For instance, Zha and Li [15] presented liquid-silicon, which employs the NVM-based crossbar tiles for both the sum-of-product logic and storage functions, effectively replacing the traditional CLBs, BRAMs, and routing resources. Likewise, Zidan et al. [26] proposed M-Cores, a concept that integrates the memory, analog computing, and digital computing within a fundamental tile, replacing the conventional CLBs, BRAMs, and DSPs with the M-cores array. Zhang et al. [16] adapted a typical CiM architecture to a dual-port two-bank BRAM architecture to bridge the architectural gap between the CiM and the BRAM. On the other hand, the CiM function can also be achieved by integrating the NVM-based heterogeneous blocks into the FPGA chip. For example, Ji et al. [14] introduced FPSA, an architecture that integrates the ReRAM-based crossbar blocks into the FPGA chip to realize the high-precision, high-parallelism NN computation. Moreover, they proposed a spatial-to-temporal mapper to map the NN model to the CiM blocks. These works demonstrate the potential of combining FPGAs with CiM, where the high programmability and customizable processing capacity of FPGAs are melded with the high parallel and low power processing advantages of

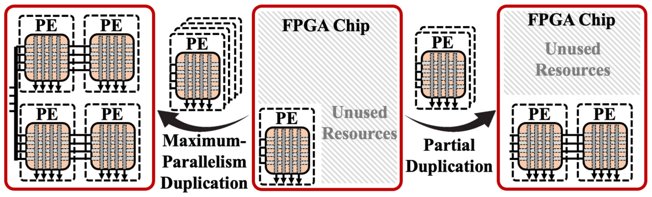

Fig. 6. Two types of PE duplication: MPD and PD.

CiM. These works significantly enhance the data processing speed and efficiency, while reducing the latency, which is particularly beneficial for the big data and high-performance computing tasks.

C. Run Time Switching on NV-FPGA

Run time switching technology is extensively employed in NV-FPGAs. Huai et al. [5] and Zhang et al. [27] introduced a run time reconfiguration mechanism to distribute application writes on the NVM-based BRAMs for the purpose of wear leveling. Subsequently, Zhang et al. [28] developed a solution for configuration switching and run time data reserving. These approaches generate multiple configuration files during the offline stage and reconfigure them during the run time stage to achieve wear leveling. As a result, some studies have integrated considerations of synthesis time, and reconfiguration cost into the configuration file generation process. Xue et al. [29] proposed an algorithm to maximize the routing path reuse, with the aim of reducing the write load to the NVM cells and improving the reconfiguration efficiency. Zhao et al. [30] introduced a correlation-guided placement approach to accelerate the configuration file list generation processing.

In this article, we utilize the run time switching technology feature to implement the oversized NNs on the CiM-equipped NV-FPGAs. Compared to the related works, our focus lies on how to achieve higher performance under the resourced limited conditions, by simultaneously considering the execution efficiency and switching cost.

III. Motivation

For the oversized NN inference, we segment the NN model into multiple tasks, each encompassing the implementation of its respective PE. These tasks are switched at run time through reprogramming. Different tasks may have different PE sizes. It is possible to increase task parallelism by duplicating PEs, especially for the tasks with smaller PE sizes. Fig. 6 depicts two types of PE duplication: 1) maximum-parallelism duplication (MPD) and 2) partial duplication (PD). MPD duplicates PEs as much as possible to achieve the highest parallelism, aiming for the reduced execution time. However, MPD increases the switching time due to the need for reprogramming a larger amount of data, potentially surpassing the benefits derived from the reduced execution time. PD takes into account the task switching time cost to determine the number of duplications, which will be elaborated in Section IV-B1.

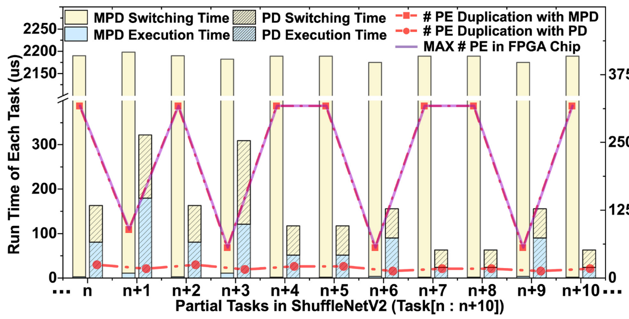

Fig. 7. Run time with different duplication settings.

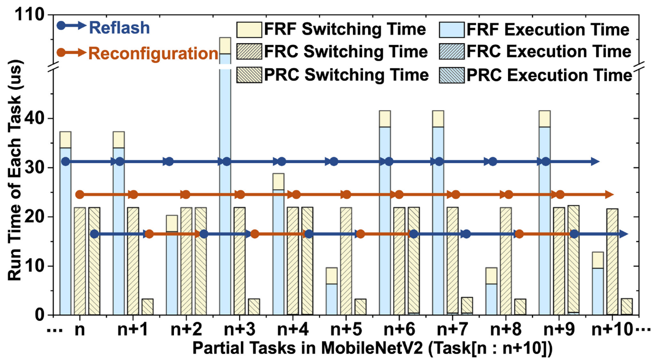

Fig. 8. Run time with different task switching decisions.

Fig. 7 demonstrates the effect of varying PE duplication numbers on the run time. The purple line represents the maximum number of PEs that can be deployed within the available FPGA resources. Due to the different PE sizes of each task, the maximum number is also different. As MPD aims to achieve the greatest parallelism, it aligns with the available maximum number. MPD achieves shorter execution time by increasing parallelism. However, due to the non-negligible task switching time, MPD may not always be the optimal choice for the task deployment. In this example, the increased task switching time to enhance the parallelism has already exceeded the benefits it offers in the execution time, resulting in a longer run time. Therefore, it is necessary to consider the switching time in the task deployment.

Furthermore, FPGA offers flexible task switching types for the NN inference, reconfiguration and reflash. In our preliminary experiments with MobileNetV2, we compare three approaches: full usage of reflash (FRF), full usage of reconfiguration (FRC), and partial usage of reconfiguration (PRC). As Fig. 8 illustrates, FRF leads to longer execution time than the others. The reason is that the reflash only updates the weight parameters and cannot modify the PE structure, necessitating an universal PE to meet the minimum requirements for the adjacent tasks. This results in reduced execution efficiency for the tasks using smaller PEs than the universal one. On the other hand, FRC restructures PE for each task, achieving higher execution performance but exhibiting longer switching time. This is because restructuring PE involves not only updating the weights of the CiM blocks but also reprogramming the connections, usually involving the larger programming data. However, since the NN inference performance is determined

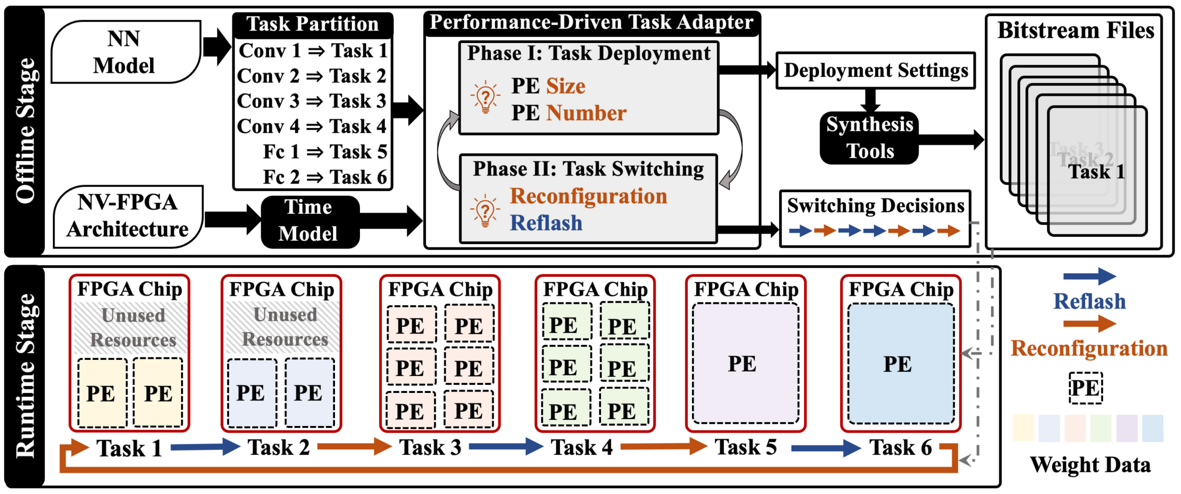

Fig. 9. Implementing oversized NN on NV-FPGAs.

by both the execution and switching time, exclusively relying on either reconfiguration or reflash is not the optimal solution. This inspires us to propose a strategic task switching approach, which utilizes PRC to reduce the overall run time.

IV. Implementing Oversized Neural Network on Nonvolatile FPGAs

In this section, we detail the implementation of the oversized NNs on the NV-FPGAs. In Section IV-A, we outline the main idea and describe the formulation of the NN inference implementation. In Section IV-B1, the task deployment approach is presented. In Section IV-B2, the task switching approach is presented.

A. Neural Network Inference Implementation on NV-FPGAs

Main Idea: CiM function blocks integrated into the NV-FPGAs allow the high-throughput NN inference. However, implementing the entire network within the FPGA is impractical, especially when faced with the limited resources or large network scales. We exploit the inherent run time reprogramming to implement the oversized NNs on the CiM-equipped NV-FPGAs by segmenting the network into multiple tasks. As shown in Fig. 9, in the offline stage, the computational graph of the NN model is partitioned into several tasks. Following this, these tasks are fed into a PD-Adapter to adapt the FPGA chip for high run time performance through the task deployment and switching. The main idea of the PD-Adapter is to achieve a shorter total run time by finding a tradeoff between the reprogramming and execution time. Task deployment focuses on optimizing the deployment settings, including the PE size and the number of PEs to improve the execution efficiency. Task switching focuses on selecting a switching type for each task to shorten the reprogramming time. The two phases alternate iteratively multiple times to pursue the minimal run time. These processes create a run time implementation solution, including the deployment settings and switching decisions. Deployment settings are used to generate corresponding bitstream files through the synthesis tools. Switching decisions consist of a series of task switching types. In the run time stage, bitstream files are programmed into the FPGA chip according to the task switching decisions.

Problem Formulation: The NN inference implementation can be formulated as a directed cycle graph G, where the vertices represent the task deployment settings and the edges indicate the task switching types. Each vertex v is assigned a value, denoting the execution time, represented as $v.val$. Each edge e has a value representing the task switching time denoted as $e.val$. The total run time of all the tasks is represented as $\sum(v.val + e.val)$. Different switching types lead to varying $e.val$, while different PE sizes and duplication numbers affect the task's execution efficiency, resulting in different $v.val$. Specifically, the v and e interact with each other. Reconfiguration reconstructs the CiM blocks and routing resources to match the PE requirement of the task, usually enhancing the execution efficiency and thus reducing the execution time, which means a small $v.val$. However, due to the changed connections in CiM by reconfiguration, programming the FPGA chip requires a longer time, resulting in a large $e.val$. Conversely, reflash only updates the weight parameters and cannot modify the PE structure, thus requiring an universal PE to meet the minimum requirements for the adjacent tasks. This results in reduced execution efficiency for tasks using smaller PEs than the universal one, thereby causing a larger $v.val$. As shown in (1), we identify suitable deployment settings v and a task switching type e to minimize the overall run time of all the tasks

$$\begin{aligned}&\min \left(\sum (v.val + e.val)\right)\\ &\text{s.t.}\quad v, e \in G. \end{aligned}\tag{1}$$

Time Model: Time models are employed to calculate the values of the vertices and the edges, which represent the execution time ($v.val$) and the reprogramming time ($e.val$), respectively.

The execution time model as shown in (2), involves the following components. *ReqCom* represents the number of times the PE being called in the task. For example, in CNNs, this corresponds to the sliding window movements of the kernel on the feature map; in the recurrent NNs, it relates to the sequence window movements; and in the attention mechanism, it pertains to the focus area movements. Para indicates the task's PE parallelism, which is represented by the number of duplications due to the lack of the data dependencies within the task. *Freq* denotes the operating frequency of the task, as reported by the EDA tool following logic and physical synthesis flow. A higher *Freq* signifies faster processing speed, resulting in a shorter execution time. N_{Cyc} denotes the number of clock cycles required for a single CiM operation

$$\text{Time}_{\text{Exe}} = \left\lceil \frac{N_{\text{ReqCom}}}{\text{Para}} \right\rceil * \frac{1}{\text{Freq}} * N_{\text{Cyc}}. \tag{2}$$

The task switching time model as shown in (3) and (4), calculates the switching time for reflash and reconfiguration types. Both reflash and reconfiguration require the reprogramming of corresponding information, i.e., the reflash and reconfiguration files into the FPGA chip. Therefore, the numerator represents the data size that needs to be reprogrammed, while the denominator ($\text{Speed}_{\text{Prog}}$) represents the reprogramming speed.

For reflash, only the weight in the CiM block needs to be updated. The time model can be formulated according to (3). Size indicates the number of CiM blocks required by a PE. Para represents the parallelism, which indicates the number of PE blocks that can be duplicated. Size $*$ Para denotes the required number of CiM blocks by a task. M_{CiM} represents the memory size of a single CiM block.

For reconfiguration, as the PE is restructured, the associated CLB, routing resources, and BRAM resources need to be reprogrammed. In typical island-style FPGAs, the heterogeneous modules are interspersed among the CLBs and routing resources, demonstrating a linear relationship in quantity. As shown in (4), M_{CLB}, M_{SB}, M_{CB}, and M_{BRAM} represent the memory sizes of CLB, SB, CB, and BRAM, respectively. We utilize the preset parameters α, β, γ, and δ to quantify the relationship with the number of CiM blocks

$$\text{Time}_{\text{Rf}} = \frac{\text{Size} * \text{Para} * M_{\text{CiM}}}{\text{Speed}_{\text{Prog}}} \tag{3}$$

$$\text{Time}_{\text{Rc}} = \frac{\text{Size} * \text{Para} * (M_{\text{CiM}} + \alpha M_{\text{CLB}} + \beta M_{SB} + \gamma M_{CB} + \delta M_{\text{BRAM}})}{\text{Speed}_{\text{Prog}}}. \tag{4}$$

B. Performance-Driven Task Adapter

The PD-Adapter is composed of the task deployment and switching phases. In the task deployment phase, the focus is on optimizing the deployment settings, including the PE size and the number of PEs as detailed in Section IV-B1. In the task switching phase, the focus is on selecting a switching type for each task to shorten the run time as detailed in Section IV-B2. The two phases alternate iteratively to optimize and achieve the minimal run time.

1) Task Deployment: Task deployment is employed to determine the optimal deployment settings for the given task switching decisions. Each task is sequentially switched and executed on the FPGA chip during the run time stage. In the task list, the PE structure changes following a task switch via reconfiguration, while reflash only updates the weight parameters. Therefore, adjacent tasks switched by reflash share the same deployment setting. As illustrated in Fig. 10, we group the task list based on the boundaries defined by the tasks with reconfiguration. Within a task group, the first task is reconfigured, while the switching type of the remaining tasks is reflash. Consequently, tasks within a group share the same deployment setting. Due to the varying computational demands of different tasks, different deployment settings can significantly affect the efficiency of the task execution. Therefore, a task deployment strategy is proposed, aiming for higher execution efficiency while considering the switching time by determining the PE deployment settings. The PE size indicates the required number of CiM blocks. An insufficient PE size will lead to mapping failure, while a large PE size could result in resource wastage. To ensure that all the tasks within the group are successfully mapped with minimal resource consumption, the width and height of the task deployment, i.e., $\text{Size}_{\text{width}}$ and $\text{Size}_{\text{height}}$ are set to the maximum values of the width and height of each task in the group. The PE size is set according to (5), G_{begin} represents the first task in the group, and G_{end} denotes the final task within the group

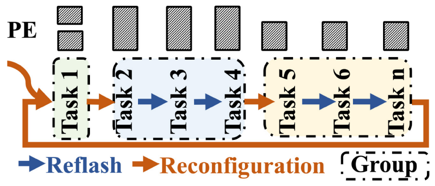

Fig. 10. Task deployment group.

$$\begin{aligned}\text{Size}_{\text{width}} &= \max\left(\text{Task}\left[G_{\text{begin}} : G_{\text{end}}\right].\text{width}\right)\\ \text{Size}_{\text{height}} &= \max\left(\text{Task}\left[G_{\text{begin}} : G_{\text{end}}\right].\text{height}\right).\end{aligned} \tag{5}$$

The number of duplications is an important parameter to consider when deploying a task, as it determines the parallelism of the task. The size of the feature map dictates the computational requirements of each task. When redundant FPGA resources are available, higher parallelism can be realized by duplicating PEs, consequently shortening the execution time. However, an increased duplication number necessitates additional resources as demonstrated in (3) and (4), thereby increasing the reprogramming time. As demonstrated in (6), determining the number of PE duplications is formulated as an integer programming problem. In this model, Para signifies the task parallelism, corresponding to the number of PE duplications. All the tasks in the group accumulate the execution time. Since, the first task uses the reconfiguration type, the calculation of Time_{Rf} begins with the second task in the group, following G_{begin}. The constraint's lower bound is 1, indicative of the minimum requirement that at least one PE is deployed, while the upper limit corresponds to the maximum number of PEs the FPGA architecture can accommodate. By solving (6), the optimal value for Para is determined, yielding the shortest total time under the current schedule while adhering to the constraints

$$\min \left(\sum_{i=G_{\text{begin}}}^{G_{\text{end}}} \text{Time}_{\text{Exe}}[i] + \sum_{i=G_{\text{begin}+1}}^{G_{\text{end}}} \text{Time}_{\text{Rf}}[i] + \text{Time}_{Rc} \right)$$
$$\text{s.t. Para} \in \left[1 : \left\lfloor \frac{\text{Res}_{\text{Tol}}}{\text{Res}_{\text{PE}}} \right\rfloor \right]. \quad (6)$$

2) Task Switching: In NV-FPGAs, the task switching can be achieved by reflash or reconfiguration. The amount of data that needs to be reprogrammed for these switching types is different, resulting in different switching time as demonstrated by (3) and (4). As shown in Fig. 10, the change in the switching type can also lead to changes in the group, thereby affecting the execution time. For example, when there is a significant difference in PE size between the adjacent tasks, reconfiguring can create a perfect match for the PEs of these tasks, thereby increasing the execution efficiency. However, this approach can lead to a potentially large switching time. We can flexibly make the switching type choice between reflash and reconfiguration, aiming to reduce the overall run time. The decision-making problem has a complexity of $(m \cdot 2)^n$, where m denotes the number of deployment settings each task can support, 2 represents the two types of switching, and n is the number of tasks. Utilizing the simulated annealing (SA) [31] method, we can find an optimal schedule with a short total run time for this problem. SA is a heuristic algorithm that employs a probabilistic acceptance mechanism and uses a random search to explore different solutions, gradually converging on the optimal solution.

As shown in Algorithm 1, the algorithm explores the optimal solution by maintaining the two lists, *dList* and *sList*, and continuously updates the information in these lists. *sList* records the switching decisions, i.e., the switching type between the tasks. *dList* records the task's deployment settings, and *dList* is updated in the manner mentioned in the task deployment phase. During the run time stage, *task*[*i*] is deployed according to the setting in *dList*[*i*], and switches to the next task using the switching type in *sList*[*i*].

At the beginning, the algorithm creates *dList* and *sList*, and randomly initializes them (lines 1–6). afterward, the algorithm enters the exploration-and-evaluation stage (lines 11–42). In this stage, the algorithm makes task deployment decisions based on the current *sList* and the corresponding *dList*. The current *sList* and *dList* are then evaluated to calculate the time cost (lines 13–17). The algorithm calculates the execution time (line 14) by (2) and computes the switching time (line 15) by (3) and (4).

Subsequently, a switching type is randomly selected from *sList* for reassignment to generate a new solution, labeled as $sList_{\text{Try}}$ (lines 19–28). In the process of generating the $sList_{\text{Try}}$, we add a series of constraints. There are some tasks that bring benefits or penalties when combined with each other. Therefore, we implement these constraints through a combo check process (lines 23 and 24). We combine these tasks to form a combo that adheres to specific beneficial or detrimental patterns. Combinations that yield benefits are termed affinitive combo, while those that incur penalties are called antagonistic combo.

Algorithm 1 Task Switching Algorithm

```
Require: Task List tList, Initial Temperature $T_0$, Final
    Temperature $T_f$, Cooling Rate $\gamma$;
Ensure: Switching List sList, Deployment List dList;
 1: Build Switching List sList and dList with tList;
 2: for $i = 0$ to $len(tList) - 1$ do
 3:   Randomly initial a switching type for sList[i];
 4:   Randomly initial a deployment setting for dList[i];
 5: end for
 6: $T = T_0$;
 7: for $i = 0$ to $len(tList) - 1$ do
 8:   Find the Affinitive Combo for tList[i];
 9:   Find the Antagonistic Combo for tList[i];
10: end for
11: while $T > T_f$ do
12:   $dList = Deployer(sList)$;
13:   for $i = 0$ to $len(tList) - 1$ do
14:     $tList[i].T_{Exe} = CalT_{Exe}(dList[i], sList[i])$;
15:     $tList[i].T_{Sw} = CalT_{Sw}(dList[i], sList[i])$;
16:   end for
17:   $Cost_{Cur} = \sum_{i=0}^{len(tList)} (tList[i].T_{Exe} + tList[i].T_{Sw})$;
18:   $sList_{Try} = sList$;
19:   while true do
20:     $sList_{Tmp} = sList_{Try}$;
21:     Randomly select an index of $sList_{Tmp}$;
22:     Change the State of $sList_{Tmp}[index]$;
23:     $sList_{Tmp} \Leftarrow$ Affinitive Combo of $tList[index]$;
24:     $Flag_{Anta} = AntaComboCheck(sList_{Tmp})$;
25:     if $Flag_{Anta} == $ "Pass" then
26:       $sList_{Try} \Leftarrow sList_{Tmp}$;
27:       break;
28:     end if
29:   end while
30:   $dList_{Try} = Deployer(sList_{Try})$;
31:   for $i = 0$ to $len(tList) - 1$ do
32:     $tList_{Try}[i].T_{Exe} = CalT_{Exe}(dList_{Try}[i], sList_{Try}[i])$;
33:     $tList_{Try}[i].T_{Sw} = CalT_{Sw}(dList_{Try}[i], sList_{Try}[i])$;
34:   end for
35:   $Cost_{Try} = \sum_{i=0}^{len(tList)} (tList_{Try}[i].T_{Exe} + tList_{Try}[i].T_{Sw})$;
36:   $\Delta Cost = Cost_{Try} - Cost_{Cur}$;
37:   if $Random(0, 1) < \exp(-\Delta Cost/T)$ then
38:     $sList \Leftarrow sList_{Try}$;
39:     $dList \Leftarrow dList_{Try}$;
40:   end if
41:   $T = \gamma T$;
42: end while
43: return sList, dList;
```

Fig. 11(a) shows the mode of affinitive combo. For some NN models, such as VGG, in order to maintain the consistency of the kernel's feature extraction method among different layers, some adjacent layers have the same kernel size. This results in two adjacent tasks having the same PE size. Combo tasks that require the same PE size are named affinitive combo. Within affinitive combos, all tasks except the first are set to switch via reflash to minimize the switching time. For the first

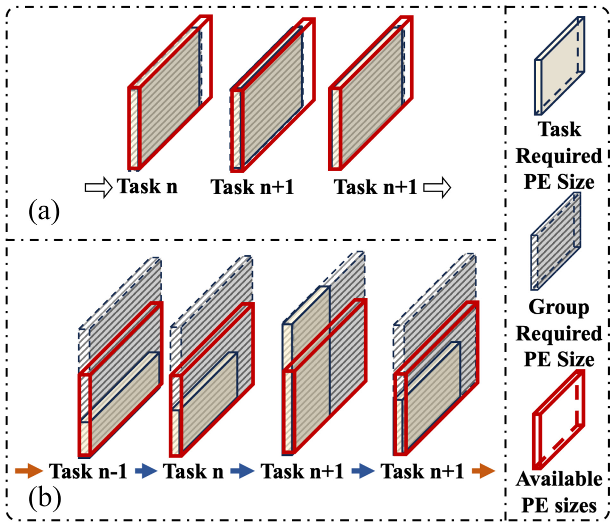

Fig. 11. Pattern of affinitive combo and antagonistic combo. (a) Pattern of affinitive combo. (b) Pattern of antagonistic combo.

task in an affinitive combo, the switching type, either reflash or reconfiguration is determined by the algorithm.

Fig. 11(b) shows the pattern of an antagonistic combo. Within a group, tasks are switched using reflash. Therefore, all the tasks in a group share the same deployment setting. According to (5), the determined PE size must meet the PE requirements of all the tasks in the group. However, shared PE size may exceed the available resources. For example, resource shortages can occur when grouping together a task with a large width and a task with a large height. We define an antagonistic combo as a situation where the grouping of the tasks necessitates an oversized PE.

The algorithm looks for the affinitive and antagonistic combo for each task (lines 7 and 10). In the process of generating $sList_{\mathrm{Try}}$, the algorithm randomly changes the reprogramming type of the *index*th task in the temporary switching list $sList_{Tmp}$ to produce an adjacent switching list, $sList_{\mathrm{Try}}$ (lines 20–22). To ensure $sList_{\mathrm{Try}}$ meets the combo constraint, first, we change all the *sList* entries corresponding to the affinitive combo of the current position to the reflash type (line 23). Subsequently, an antagonistic combo check is performed (lines 24–28). The *AntaComboCheck*() function checks whether there is an antagonistic combo in $sList_{Tmp}$, and this process is repeated until $sList_{Tmp}$ no longer contains any antagonistic combo.

Afterward, the new switching list $sList_{\mathrm{Try}}$ invokes the task deployment (line 12). By solving (6) and (5), the related optimal deployment settings list $dList_{\mathrm{Try}}$ is obtained. The algorithm evaluates the execution and switching time using $sList_{\mathrm{Try}}$ and $dList_{\mathrm{Try}}$, and calculates the cost of the new solution (lines 31 and 34). A negative $\Delta Cost$ indicates that $sList_{Tmp}$ has a shorter run time, and the algorithm will accept this trial. Conversely, a positive ΔCost means that $sList_{Cur}$ is worse than the current solution, and the algorithm will accept it with a probability $P = e^{(-\Delta \mathrm{Cost}/T)}$ (line 37). This probability P decreases as the number of iterations of the algorithm increases. Initially, the algorithm tolerates bad solutions by allowing a worse *sList*, which aids in avoiding premature entrapment in the local optima. As the temperature progressively decreases, the algorithm's acceptance of inferior solutions diminishes, ultimately leading to the identification of the global optimum. The efficacy of this method hinges on striking a balance between the exploration and precise optimization.

TABLE I
Parameters of NV-FPGA Architecture

Parameter	Value
Number of BLEs Per Cluster	10
Channel Width	150
Wire Segment Length	4, 16
Number of Cluster Inputs	60
Number of Cluster Outputs	40
LUT Size	6-LUT (5-LUTx2)
Switch Block Flexibility	3
CiM Block Type	BRAM Integration [16]
CiM Column	64
CiM Row	64
Programming Speed	400 MB/s [32]

V. Evaluation

In this section, we introduce experimental setup, report evaluation results, and give discussions.

A. Experiment Setup

We have implemented the proposed PD-Adapter and integrated it into the open-source FPGA toolchain VTR [33]. We utilize the Pytorch [34] tool for the model description and the task segmentation, and employ Yosys [35] for logic synthesis. The architectural parameters of the NV-FPGA can be found in Table I. Computational BRAM is utilized as the CiM Block, which is based on the Altera Stratix-IV-like device,[1] with a crossbar size of 64×64. In the foundational experiments presented in Section V-B, the FPGA size is set to the minimum necessary for deploying any layer of the NN model. The Xilinx internal configuration access port (ICAP) technique is employed for the partial reprogramming to achieve the task switching. The bitstream files are stored in off-chip double-data-rate (DDR) memory and reprogrammed via the Xilinx AXI HWICAP interface [32]. Details of the benchmark NN models are provided in Table II, with the benchmarks encompassing NNs ranging from 1M to 100M in terms of the weight count. To validate the generalizability of our method across different resource availabilities, we conducted evaluations on the FPGAs of varying scales as referenced in Section V-C.

We compare the following implementations.

1) *Baseline:* Task switching types are set to reflash. The size of the deployment is set to satisfy the minimum size requirement of any layer. No additional decisions are made regarding the task switching or task deployment.

[1]The integration of computational BRAM is realized by modifying the architecture definition file in line with k6FracN10LB_mem20K_complexDSP_customSB_22nm.xml [16], [33]. The design of the crossbar structure and delay information in the CiM block is based on MNSIM [36].

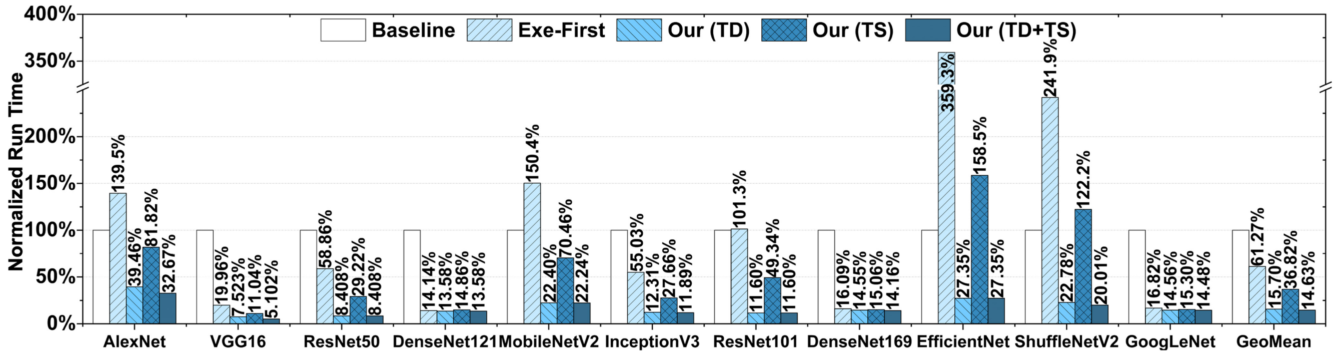

Fig. 12. Run time with all tested implementations.

TABLE II
INFORMATION OF BENCHMARKS

NN Model	Domain	# of Weights	# of OPs*
AlexNet	Image Recognition	60M	1.5G
VGG16	Image Recognition	138M	15.3G
ResNet50	Image Recognition	25M	3.8G
DenseNet121	Image Recognition	8M	2.8G
MobileNetV2	Image Processing	3.5M	0.3G
InceptionV3	Image Recognition	23M	5.7G
ResNet101	Image Recognition	44M	7.8G
DenseNet169	Image Recognition	14M	3.5G
EfficientNet	Image Processing	5.3M	0.4G
ShufflenetV2	Image Processing	1.4M	0.05G
GoogLeNet	Image Recognition	6M	1.5G

* In ImageNet

2) *Execution-Time-First Policy (Exe-First):* Task switching types are set to reconfiguration. To optimize execution time, it adopts MPD.

3) *Our (TD):* This approach exclusively implements task deployment. To emphasize the optimization impact of these deployment settings, all the task switching types are set to reconfiguration.

4) *Our (TS):* This approach exclusively implements task switching. For a clearer illustration of their optimization impact, all the task deployment settings adhere to MPD.

5) *Our (TD+TS):* This approach realizes PD-Adapter, optimizing both the task switching and deployment collaboratively to pursue a shorter run time.

B. Evaluation Results

Fig. 12 displays the run time of all the tested implementations in all the benchmarks with the data normalized to those of the *Baseline*. Although *Exe-First* decreases the geometric mean of run time by 38% compared to the *Baseline*, nearly half of the benchmarks exhibit poor performance. This is attributed to the fact that these NN models have similar PE requirements in consecutive layers, or some tasks exhibit lower computational demands, making the performance benefits of using reconfiguration during the task switching less apparent. Moreover, reconfiguration results in increased task switching time due to the reprogramming of routing resources and configurable resources related to the PE structure. *Our (TD)* in comparison with *Baseline* and *Exe-First*, achieves a run time reduction of 38.73% and 74.38%, respectively. This is attributed to the fact that *Our (TD)* utilizes a task deployment to strategically determine the number of PE duplications, thereby avoiding a significant increase in switching time arising from the pursuit of the execution efficiency. *Our (TS)*, considering the execution time, intelligently selects task switching types. As shown in Table III, it employs reconfiguration to enhance the execution efficiency, thereby reducing run time by 63.18% compared to *Baseline*. *Our (TD+TS)* considering both the task switching and the task deployment, achieves a run time reduction of 85.37% and 76.12% compared to the *Baseline* and *Exe-First*, respectively. As shown in Table III, although both *Our (TS)* and *Our (TD+TS)* employ the task switching, the differences in the task deployment lead to significant differences in their choice of switching types. They are collaboratively optimized to achieve better performance.

To provide further details, we use AlexNet as an illustrative example to demonstrate its specific reduction in run time as depicted in Fig. 13. The AlexNet model is segmented into eight tasks according to its layers, and we present the execution time and the task switching time of each task. Each task in the *Baseline* exhibits the same switching time, as all the task switching types are set to reflash. To ensure all the tasks can be mapped without restructuring PE, the PE size is set to the maximum height and width among the tasks, resulting in a noticeable waste of resources. It is evident that Tasks 1–4 have a markedly long execution time. Compared with *Exe-First*, *Our (TS)* widely selects switching types. In cases, such as Tasks 2 to 3, Tasks 4 to 5, Tasks 6 to 7, and Tasks 8 to 1, *Our (TS)* selects reflash. Although reflash incurs a reduction in execution performance, its advantages in reducing the task switching time are more pronounced, leading to a shorter total run time. Furthermore, by employing the task deployment, *Our (TD)* makes a tradeoff between the parallelism and switching time. Therefore, the total execution time is longer compared to the *Exe-First* policy and *Our (TS)*. However, this shortening of the switching time leads to greater performance improvement. *Our (TD+TS)* utilizes both the task deployment and switching collaboratively.

C. Discussions

1) Run Time With Different Datasets: Across various datasets, the computational requirements of NNs differ significantly. This variation is due to the different input sizes, where the smaller input sizes lead to less computation for

TABLE III
NUMBERS OF REPROGRAMMING IN ALL TESTED IMPLEMENTATIONS. (RC: RECONFIGURATION, RF: REFLASH)

	AlexNet		VGG16		ResNet50		DenseNet121		MobileNetV2		InceptionV3		ResNet101		DenseNet169		EfficientNet		ShuffleNetV2		GoogleLeNet	
	RC	RF	RC	RF	RC	RF	RC	RF	RC	RF	RC	RF	RC	RF	RC	RF	RC	RF	RC	RF	RC	RF
Baseline	-	8	-	16	-	52	-	128	-	51	-	98	-	101	-	160	-	79	-	56	-	16
Exe-First	8	-	16	-	52	-	128	-	51	-	98	-	101	-	160	-	79	-	56	-	16	-
Our(TD)	8	-	16	-	52	-	128	-	51	-	98	-	101	-	160	-	79	-	56	-	16	-
Our(TS)	4	4	6	10	20	32	54	74	18	33	46	52	47	54	52	108	18	61	18	38	7	9
Our(TD+TS)	5	3	6	10	51	1	62	66	26	25	42	56	89	9	87	73	37	42	16	40	6	10

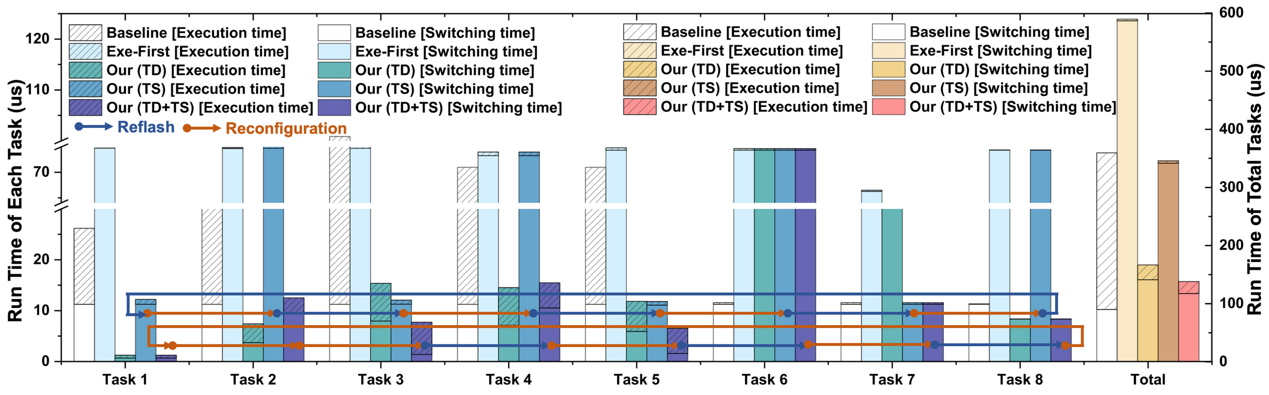

Fig. 13. Detailed run time of AlexNet.

a single inference, significantly increasing the proportion of the task switching time. Table IV illustrates the run time of different policies on the MNIST [37], COCO-Medium [38], and ImageNet [39] datasets. Input sizes for these datasets are as follows: MNIST (28, 28), COCO-Medium (96, 96), and ImageNet (224, 224). Observations indicate that as the input size increases, the performance improvement of *Exe-First* and *Our* becomes more pronounced. This occurs because an increase in size results in higher computation for each layer, thus enhancing the benefits of reconfiguration in terms of the execution time, potentially outweighing the switching costs. Conversely, when computational demand decreases and the number of switches remains constant, the overall proportion of switching time increases, thus accentuating the advantage of reflash. *Our (TD+TS)* accounting for both the computational demands and the switching time, exhibits stable performance across various datasets.

2) Run Time With Different FPGA Scales: Different networks exhibit varying computational resource demands, and we accordingly allocate FPGA size based on these requirements. We use "scale" to represent the relative size of physical and logical resources, and the layout size is shown in Table V.[2] We conduct tests under different FPGA scales. The $1\times$ scale is defined as the size necessary to meet the needs of all the layers with reflash, as the CiM block constitutes the resource bottleneck in the NN inference. Consequently, we evaluate the run time of *Baseline*, *Exe-First*, and *Our* under the FPGA scales of 1.2, 1.5, 2, and $4\times$ specifically for the CiM block. All the benchmark results are calculated for the geometric mean and are normalized to the $1\times$ scale of the *Baseline*.

Fig. 14 shows the run time under different FPGA scales. The results indicate that as the FPGA scale increases, *Exe-First* can duplicate more PEs, thereby shortening the run time. However, at 1.2 and $1.5\times$ scales, the run time remains identical to that at $1\times$. This occurs since *Baseline* is limited to updating the weight data of PEs during operation and cannot modify the size of PEs, while the CiM resources increase by $\lfloor 1.2 \rfloor\times$ and $\lfloor 1.5 \rfloor\times$. *Our (TD+TS)* balancing execution time and switching, can duplicate more PEs when computational demands are high and reduce the number of duplications or choose reflash type when the computational demands are low, thus making reasonable use of the FPGA resources. As a result, it achieves better performance as the scale increases.

3) Lifetime Evaluation: The proposed run time switching mechanism involves frequent task switching during the run time stage, which could potentially lead to the lifetime issues. We evaluate the lifetime of all the implementations using a round-Robin approach for wear leveling. For each reconfiguration, the PEs are reassigned to another adjacent CiM blocks to avoid excessive use of certain CiM blocks. As shown in Fig. 15, compared to *Baseline* and *Exe-First*, *Our (TD+TS)* achieves a higher lifetime. For *Baseline* and *Exe-First*, the PE duplication in the deployment setting is set to MPD. Consequently, the unused resources are insufficient for effective wear leveling. Due to the fact that *Our (TD+TS)* does not occupy all the CiM resources at PE duplication for the tasks with small computational requirements, it reduces the number of writes. At scales of 1.2 and $1.5\times$, the excess resources are insufficient to allow the *Baseline* to duplicate PEs, thus achieving a lifetime increase of 1.2 and $1.5\times$, respectively. Furthermore, we also validated another *Baseline*-based policy incorporating lifetime considerations, namely *Baseline-OPL*, which does not perform the PE duplication to reserve more spare resources to realize wear leveling. However, its lifetime improvement was still inferior to *Our (TD+TS)*.

4) Comparison With DSP-Based PE Implementation: The proposed run time task switching mechanism utilizes the partial programming technique to reprogram the weight

[2]Under the k6FracN10LB_mem20K_complexDSP_customSB_22nm.xml architecture [16], [33], the width is the same as the height.

TABLE IV
NORMALIZED RUN TIME WITH DIFFERENT DATESETS

	MNIST			COCO-Medium			ImageNet		
	Baseline	Exe-First	Our (TD+TS)	Baseline	Exe-First	Our (TD+TS)	Baseline	Exe-First	Our (TD+TS)
AlexNet	100.00%	374.10%	63.64%	100.00%	292.66%	56.25%	100.00%	139.48%	32.67%
VGG16	100.00%	351.95%	35.02%	100.00%	88.91%	13.92%	100.00%	19.96%	5.10%
ResNet50	100.00%	520.18%	29.96%	100.00%	222.47%	18.85%	100.00%	58.86%	8.41%
DenseNet121	100.00%	257.98%	56.27%	100.00%	40.07%	24.70%	100.00%	14.15%	13.58%
MobileNetV2	100.00%	554.10%	36.05%	100.00%	379.67%	31.82%	100.00%	150.35%	22.24%
InceptionV3	100.00%	55.03%	11.89%	100.00%	55.03%	11.76%	100.00%	55.03%	11.89%
ResNet101	100.00%	578.79%	28.52%	100.00%	321.04%	21.73%	100.00%	101.34%	11.60%
DenseNet169	100.00%	302.20%	51.87%	100.00%	50.66%	25.48%	100.00%	16.09%	14.16%
EfficientNet	100.00%	614.40%	34.12%	100.00%	548.37%	32.36%	100.00%	359.34%	27.35%
ShuffleNetV2	100.00%	611.55%	23.00%	100.00%	486.42%	25.05%	100.00%	241.94%	20.01%
GoogLeNet	100.00%	283.43%	55.98%	100.00%	48.14%	25.66%	100.00%	16.82%	14.48%
GeoMean	**100.00%**	**351.23%**	**35.26%**	**100.00%**	**152.16%**	**24.07%**	**100.00%**	**61.27%**	**14.63%**

TABLE V
FPGA LAYOUT SIZE FOR DIFFERENT NN MODELS

FPGA Scale	1x	1.2x	1.5x	2x	4x
AlexNet	410	448	502	580	818
VGG16	676	737	826	952	-
ResNet50	206	226	251	292	410
DenseNet121	76	82	89	104	148
MobileNetV2	221	244	274	314	442
InceptionV3	208	227	256	293	413
ResNet101	206	226	251	292	410
DenseNet169	88	98	107	124	176
EfficientNet	406	442	496	572	808
ShuffleNetV2	70	76	86	100	137
GoogLeNet	100	107	119	137	196

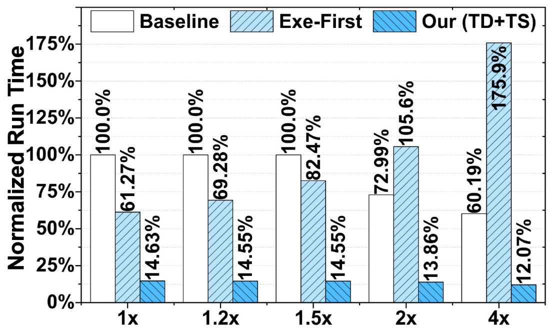

Fig. 14. Run time with different FPGA scales.

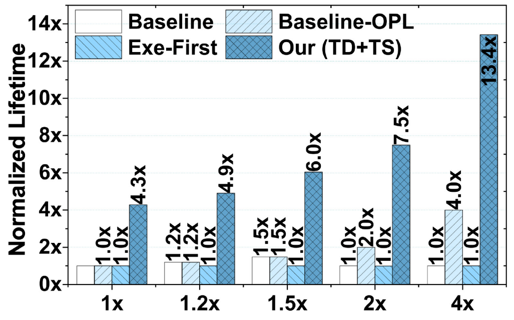

Fig. 15. Lifetime evaluation.

data and PE structure information multiple times, which introduces the reprogramming time costs. Additionally, the data throughput of the DDR memory interface with the partial programming technique is lower than that of the standard DDR memory interfaces, which might affect the performance [32], [40]. For example, the data throughput of AXI HWICAP is 400 MB/s [32], while the standard DDR memory interface can reach up to 2226 MB/s [40]. Therefore, we set up a control group called no reprogramming (NRP), which uses DSPs for PE instead of CiM blocks. *NRP* schedules the NN operations sequentially without reprogramming during the run time, allowing the data to be loaded with higher throughput using the standard DDR memory interface. As shown in Fig. 16, compared to *NRP*, *Our (TD+TS)* has a shorter run time. Although *NRP* does not require reprogramming and has higher data throughput, *Our (TD+TS)* achieves a shorter execution time due to the high computational density of CiM blocks and their efficient utilization. This advantage of *Our (TD+TS)* becomes more pronounced as the computational load increases.

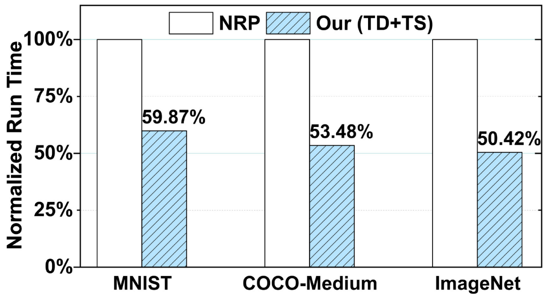

Fig. 16. Run time with different PE implementations.

5) Run Time With Different Task Partition Methods: The PD-Adapter supports various task partition schemes. In the base experiment, tasks are divided by the layer, assuming FPGA resources can meet the needs of any single layer. As the network scale increases, FPGA resources may become insufficient, necessitating a finer-grained task partition method. We design the hybrid-granularity partition (HGP), which partitions tasks at both the layer and the channel levels. HGP divides the largest layouts, which are the computational bottlenecks, into multiple subtasks at the channel level to eliminate the resource constraints. As shown in Fig. 17, *Our (TD+TS)* still shortens the run time the most. It is worth noting that *Our (TD+TS)* with HGP achieves a shorter run time compared to the layer-based partitioning method, as it partitions the operations required for the large layers, which may be resource demanding but have low computational loads like the FC layers. This results in a slight increase in execution time but an overall shorter run time.

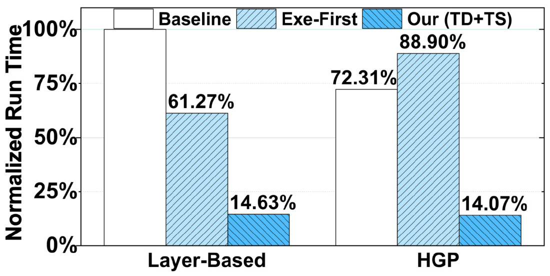

Fig. 17. Run time with different task partition methods.

VI. Conclusion

This article utilizes the inherent run time reprogramming feature of FPGA to implement oversized NNs on the CiM-equipped NV-FPGAs. A PD-Adapter is proposed, comprising the task deployment and task switching phases. In the task deployment phase, the focus is on optimizing the deployment settings. In the task switching phase, the focus is on wisely selecting a switching type for each task. These phases co-optimize task execution efficiency and task switching time cost. Finally, we have integrated the proposed PD-Adapter into an open-source toolchain for evaluation. The evaluation results demonstrate that it achieves 85.37% and 76.12% reductions in run time compared to the *Baseline* and *Exe-First*, respectively.

References

[1] Y. Chen, J. Zhao, and Y. Xie, “3D-NonFAR: Three-dimensional non-volatile FPGA architecture using phase change memory,” in *Proc. ISLPED*, 2010, pp. 55–60.

[2] H.-S. P. Wong et al., “Metal–oxide RRAM,” *Proc. IEEE*, vol. 100, no. 6, pp. 1951–1970, Jun. 2012.

[3] S. Paul, S. Mukhopadhyay, and S. Bhunia, “Hybrid CMOS-STTRAM non-volatile FPGA: Design challenges and optimization approaches,” in *Proc. ICCAD*, 2008, pp. 589–592.

[4] A. Ahari, H. Asadi, B. Khaleghi, and M. B. Tahoori, “A power-efficient reconfigurable architecture using PCM configuration technology,” in *Proc. DATE*, 2014, pp. 1–6.

[5] S. Huai, W. Song, M. Zhao, X. Cai, and Z. Jia, “Performance-aware wear Leveling for block RAM in nonvolatile FPGAs,” in *Proc. DAC*, 2019, pp. 1–6.

[6] R. Rajaei, “Radiation-hardened design of nonvolatile MRAM-based FPGA,” *IEEE Trans. Magn.*, vol. 52, no. 10, pp. 1–10, Oct. 2016.

[7] L. Ju et al., “NVM-based FPGA block RAM with adaptive SLC-MLC conversion,” *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2661–2672, Nov. 2018.

[8] S. Yazdanshenas, K. Tatsumura, and V. Betz, “Don’t forget the memory: Automatic block RAM modelling, optimization, and architecture exploration,” in *Proc. FPGA*, 2017, pp. 115–124.

[9] H. Zheng, H. Zhang, S. Xu, F. Xu, and M. Zhao, “Adaptive mode transformation for wear leveling in nonvolatile FPGAs,” *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, vol. 41, no. 11, pp. 3591–3601, Nov. 2022.

[10] P. Chi et al., “PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory,” in *Proc. 43rd ISCA*, 2016, pp. 27–39.

[11] K. Roy, I. Chakraborty, M. Ali, A. Ankit, and A. Agrawal, “In-memory computing in emerging memory technologies for machine learning: An overview,” in *Proc. DAC*, 2020, pp. 1–6.

[12] Z. Shen, J. Wu, X. Jiang, Y. Zhang, L. Ju, and Z. Jia, “PRAP-PIM: A weight pattern reusing aware pruning method for ReRAM-based PIM DNN accelerators,” *High-Confid. Comput.*, vol. 3, no. 2, pp. 100–123, 2023.

[13] C.-X. Xue et al., “15.4 a 22nm 2Mb ReRAM compute-in-memory macro with 121-28TOPS/W for multibit MAC computing for tiny AI edge devices,” in *Proc. ISSCC*, 2020, pp. 244–246.

[14] Y. Ji, Y. Zhang, X. Xie, and S. Li, “FPSA: A full system stack solution for reconfigurable ReRAM-based NN accelerator architecture,” in *Proc. ASPLOS*, 2019, pp. 733–747.

[15] Y. Zha and J. Li, “Liquid Silicon-Monona: A reconfigurable memory-oriented computing fabric with scalable multi-context support,” in *ACM SIGPLAN Not.*, 2018, pp. 214–228.

[16] H. Zhang, M. Zhao, H. Zheng, Y. Xiong, Y. Zhang, and Z. Shen, “Towards high-throughput neural network inference with computational BRAM on nonvolatile FPGAs,” in *Proc. DATE*, 2024, pp. 1–6.

[17] M. Huang et al., “Programming and runtime support to blaze FPGA accelerator deployment at Datacenter scale,” in *Proc. SoCC*, 2016, pp. 456–469.

[18] Q. Lou, M. Zhao, L. Ju, C. J. Xue, J. Hu, and Z. Jia, “Runtime and reconfiguration dual-aware placement for SRAM-NVM hybrid FPGAs,” in *Proc. NVMSA*, 2017, pp. 1–6.

[19] I. Beretta, V. Rana, D. Atienza, and D. Sciuto, “Run-time mapping of applications on FPGA-based reconfigurable systems,” in *Proc. ISCAS*, 2010, pp. 3329–3332.

[20] S. Attia and V. Betz, “Feel free to interrupt: Safe task stopping to enable FPGA checkpointing and context switching,” *ACM Trans. Reconfig. Technol. Syst.*, vol. 13, no. 1, pp. 1–27, 2020.

[21] K. Huang, Y. Ha, R. Zhao, A. Kumar, and Y. Lian, “A low active leakage and high reliability phase change memory (PCM) based non-volatile FPGA storage element,” *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 61, no. 9, pp. 2605–2613, Sep. 2014.

[22] P.-E. Gaillardon, D. Sacchetto, S. Bobba, Y. Leblebici, and G. De Micheli, “GMS: Generic memristive structure for non-volatile FPGAs,” in *Proc. 20th VLSI-SoC*, 2012, pp. 94–98.

[23] X. Tang, G. De Micheli, and P.-E. Gaillardon, “A high-performance FPGA architecture using one-level RRAM-based multiplexers,” *IEEE Trans. Emerg. Topics Comput.*, vol. 5, no. 2, pp. 210–222, Apr.–Jun. 2017.

[24] K. Liu, M. Zhao, L. Ju, Z. Jia, J. Hu, and C. J. Xue, “Applying multiple level cell to non-volatile FPGAs,” *ACM Trans. Embed. Comput. Syst.*, vol. 19, no. 4, pp. 1–27, 2020.

[25] D. Choi, K. Choi, and J. D. Villasenor, “New non-volatile memory structures for FPGA architectures,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 7, pp. 874–881, Jul. 2008.

[26] M. A. Zidan, Y. Jeong, J. H. Shin, C. Du, Z. Zhang, and W. D. Lu, “Field-programmable crossbar array (FPCA) for reconfigurable computing,” *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 4, no. 4, pp. 698–710, Oct.–Dec. 2017.

[27] H. Zhang, H. Zheng, S. Li, Y. Zhang, M. Zhao, and X. Cai, “Lifetime improvement through adaptive reconfiguration for nonvolatile FPGAs,” *J. Syst. Archit.*, vol. 128, Jul. 2022, Art. no. 102532.

[28] H. Zhang, K. Liu, M. Zhao, Z. Shen, X. Cai, and Z. Jia, “Pearl: Performance-aware wear leveling for nonvolatile FPGAs,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 2, pp. 274–286, Feb. 2021.

[29] Y. Xue, P. Cronin, C. Yang, and J. Hu, “Routing path reuse maximization for efficient NV-FPGA reconfiguration,” in *Proc. ASP-DAC*, 2016, pp. 360–365.

[30] M. Zhao et al., “Correlation-guided placement for nonvolatile FPGAs,” in *Proc. DAC*, 2023, pp. 1–6.

[31] P. J. M. Laarhoven and E. H. L. Aarts, “Simulated annealing: theory and applications,” in *Simulated Annealing*. Dordrecht, The Netherlands: Springer, 1987.

[32] *AXI HWICAP V3.0 Product Guide (PG134)*, Xilinx Inc., San Jose, CA, USA, 2016.

[33] J. Luu et al., “VTR 7.0: Next generation architecture and CAD system for FPGAs,” *ACM Trans. Reconfig. Technol. Syst.*, vol. 7, no. 2, pp. 1–30, 2014.

[34] A. Paszke et al., “PyTorch: An imperative style, high-performance deep learning library,” in *Proc. 33rd Int. Conf. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 8026–8037.

[35] C. Wolf. “Yosys open SYnthesis suite.” 2016. [Online]. Available: https://yosyshq.net/yosys/

[36] L. Xia et al., “MNSIM: Simulation platform for memristor-based neuromorphic computing system,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 5, pp. 1009–1022, May 2018.

[37] L. Deng, “The MNIST database of handwritten digit images for machine learning research,” *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 141–142, Nov. 2012.

[38] T. Lin et al., “Microsoft COCO: Common objects in context,” 2014, *arXiv:1405.0312*.

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and Li Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *Proc. CVPR*, 2009, pp. 248–255.

[40] *LogiCORE IP Multi-Port Memory Controller (DS643)*, Xilinx Inc., San Jose, CA, USA, 2013.