Mathematical Framework for Optimizing Crossbar Allocation for ReRAM-based CNN Accelerators

WANQIAN LI, Institute of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences, China
YINHE HAN and XIAOMING CHEN, Institute of Computing Technology, Chinese Academy of Sciences, China

The resistive random-access memory (ReRAM) has widely been used to accelerate convolutional neural networks (CNNs) thanks to its analog in-memory computing capability. ReRAM crossbars not only store layers' weights, but also perform in-situ matrix-vector multiplications which are core operations of CNNs. To boost the performance of ReRAM-based CNN accelerators, crossbars can be duplicated to explore more intra-layer parallelism. The crossbar allocation scheme can significantly influence both the computing throughput and bandwidth requirements of ReRAM-based CNN accelerators. Under the resource constraints (i.e., crossbars and memory bandwidths), how to find the optimal number of crossbars for each layer to maximize the inference performance for an entire CNN is an unsolved problem. In this work, we find the optimal crossbar allocation scheme by mathematically modeling the problem as a constrained optimization problem and solving it with a dynamic programming based solver. Experiments demonstrate that our model for CNN inference time is almost precise, and the proposed framework can obtain solutions with near-optimal inference time. We also emphasize that communication (i.e., data access) is an important factor and must also be considered when determining the optimal crossbar allocation scheme.

CCS Concepts: • **Hardware → Emerging architectures**; • **Computer systems organization → Neural networks;**

Additional Key Words and Phrases: ReRAM crossbars, CNN accelerator, mathematical framework, crossbar allocation

ACM Reference format:
Wanqian Li, Yinhe Han, and Xiaoming Chen. 2023. Mathematical Framework for Optimizing Crossbar Allocation for ReRAM-based CNN Accelerators. *ACM Trans. Des. Autom. Electron. Syst.* 29, 1, Article 21 (December 2023), 24 pages.
https://doi.org/10.1145/3631523

This work was supported by National Key R&D Program of China under Grant 2018YFA0701500, by National Natural Science Foundation of China under Grants 62122076, 61834006 and 62025404, by Key Research Program of Frontier Sciences, CAS under Grant ZDBS-LY-JSC012, by Strategic Priority Research Program of CAS under Grant XDB44000000, and by the Innovation Funding of ICT, CAS under Grant E261040.
Authors' addresses: W. Li, Institute of Computing Technology, Chinese Academy of Sciences and University of Chinese Academy of Sciences, 6 Kexueyuan S Rd, Haidian District, Beijing, 100190, China; e-mail: liwanqian20s@ict.ac.cn; Y. Han and X. Chen (Corresponding author), Institute of Computing Technology, Chinese Academy of Sciences, 6 Kexueyuan S Rd, Haidian District, Beijing, 100190, China; e-mails: {yinhes, chenxiaoming}@ict.ac.cn.

1084-4309/2023/12-ART21
https://doi.org/10.1145/3631523

1 INTRODUCTION

Convolutional neural networks (CNNs) have widely been adopted in numerous practical applications, such as image segmentation, object detection, and face identification (e.g., [11, 18]). The tremendous data requirements of CNN models bring great challenges to the design of CNN accelerators. For conventional von Neumann architecture based CNN accelerators, communication between processors and memories has become the bottleneck of both performance and energy efficiency [2, 3, 5, 6, 9]. Recently, in-memory computing has emerged as a promising solution to reduce data movements between arithmetic and storage units. At the highest level, in-memory computing can be categorized into two kinds of approaches: compute-near-memory (e.g., [10, 16, 17, 19, 20, 27]) and compute-using-memory (e.g., [1, 7, 8, 15, 22–25, 28–30, 33, 34]). Compute-near-memory means moving computational resources near where data is stored, while compute-using-memory means using memory cells/arrays for both storage and computing. Compute-using-memory has widely been studied for CNN acceleration, which is also the focus of this paper.

The **resistive random-access memory (ReRAM)** is popular to build in-memory architectures owing to its merits like high parallelism and fast read speed [31, 32]. More importantly, information can be programmed into the resistance of ReRAM devices. ReRAM cells can construct a crossbar structure that can perform an analog **matrix-vector multiplication (MVM)** with $O(1)$ time complexity [12] in theory, as shown in Figure 1. This feature makes ReRAMs attractive for CNN acceleration which involves numerous **multiply-accumulate (MAC)** operations. In fact, plenty of works have demonstrated that ReRAM-based architectures can effectively accelerate CNNs with higher energy efficiency (e.g., [1, 7, 8, 15, 22–25, 28–30, 33, 34]), compared with conventional **complementary metal-oxide-semiconductor (CMOS)** based CNN accelerators. For instance, in Ref. [25], the proposed ReRAM-based architecture yields 14.8× higher throughput than CMOS-based DaDianNao [4], while energy consumption is only 18% of that of DaDianNao.

After training, the weights of an entire CNN are programmed into ReRAMs' resistance, and they are not changed during inference. ReRAM-based in-memory architectures perform computations in-situ in memory and, thus, there are plentiful computing resources. This provides an opportunity to compute all layers of a CNN concurrently in a pipelined way (like [28]) to achieve higher performance. Therefore, computing resources, namely, ReRAM crossbars, need to be allocated to all layers. The overall performance heavily depends on the resource allocation strategy. Intuitively, if we use more resources for one layer (i.e., duplicating its weights to more crossbars), the layer has more parallelism and may be computed faster. Some intuitive allocation strategies are to assign ReRAM crossbars to layers proportionally to the amount of weights or workloads of each layer. However, we will show by experiments that such straightforward ideas are not good solutions. Optimal resource allocation is a fundamental problem for ReRAM-based CNN accelerators, but it has not well been solved till now.

A few existing studies have more or less involved the resource allocation problem. They heuristically determine the number of ReRAM crossbars allocated to each layer [7, 21, 25, 28, 35]. PRIME [7] duplicates the crossbars twice to hide data access latency with a ping-pong mode. The strategy is rough and cannot fully utilize the resources. It also ignores the latency difference between communication and computation. ISAAC [25] allocates crossbars heuristically, according to the stride sizes of **convolutional (Conv)** layers. PipeLayer [28] presents the results of crossbar duplication ratios but how to determine them is completely not mentioned. Ref. [35] optimizes the resource allocation for different sized crossbars to boost the resource utilization. The solution is heuristically derived and not guaranteed to be optimal. Ref. [21] proposes a throughput-aimed resource mapping strategy for multi-modal neural networks (e.g., a neural network composed of a CNN and a recurrent neural network), which are uncommon in practice, especially running on ReRAM-based accelerators.

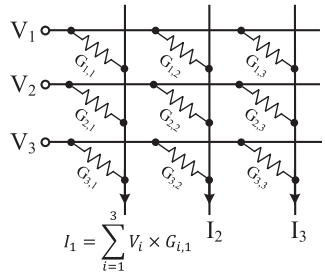


Fig. 1. ReRAM crossbar.

The resource allocation approaches in previous works are generally based on heuristic methods and are usually aimed at dedicated accelerators and conditions. For example, under the conditions where crossbar resources are scarce, allocating crossbars with the approach mentioned in ISAAC [25] will lead to the suboptimality of the accelerator performance. Our experiments demonstrate that, when the number of crossbars of ISAAC is reduced to 0.125×, the accelerator performance of accelerating VGG-A will become 0.07×, less than 0.125×. It suggests that when accelerators are computation-dominated, the performance degrades more rapidly than the reduction of computation resources, that is, the efficiency of ISAAC's resource allocation method will decline under the condition with limited resources. Moreover, existing works generally ignore the impact of data access overhead and lack consideration of the memory bandwidth. In other words, they determine the resource allocation scheme by only considering computation. In fact, as the number of crossbars increases, while the computational parallelism rises, communication burden within the accelerator will become more severe and may become the bottleneck, which cannot be ignored. How to optimally allocate resources to all layers of a CNN is still an open question. Comprehensively studying the problem is necessary.

In this work, we propose a mathematical framework to systematically solve the resource allocation problem for ReRAM-based CNN accelerators. Given a CNN model and the hardware resource constraints, the framework mathematically models the inference time as a function of crossbar allocation, and finds the optimal crossbar allocation scheme through solving a constrained problem. Compared with the previous works, our proposed resource allocation method is based on an abstract and universal performance analysis model and a dynamic programming based search strategy. It can be flexibly applied to various conditions and most ReRAM-based CNN accelerators. Near-optimal crossbar allocation schemes can be efficiently found. Besides modeling computation, we also take into account the impact of data access by adding memory bandwidth constraints to the model. The crossbar allocation approach we derived can balance the computation throughput and data access overhead across different layers. Specifically, we make the following contributions in this paper.

— We formulate a crossbar allocation problem for ReRAM-based CNN accelerators to maximize the performance. The formulated constrained optimization problem is solved by a dynamic programming based solver. Near-optimal crossbar allocation schemes can be quickly found, without time-consuming exhaustive searches.
— We mathematically model the performance of ReRAM-based CNN accelerators when considering crossbar allocation. Specifically, we first model the computational behavior by considering essential operations and building an abstract multi-level pipelined execution flow. Communication (i.e., data access) cost is further modeled by taking into account bandwidth constraints. The model can be applied to various ReRAM-based CNN accelerators. It achieves about 98% accuracy on average, compared with cycle-level behavior simulations.

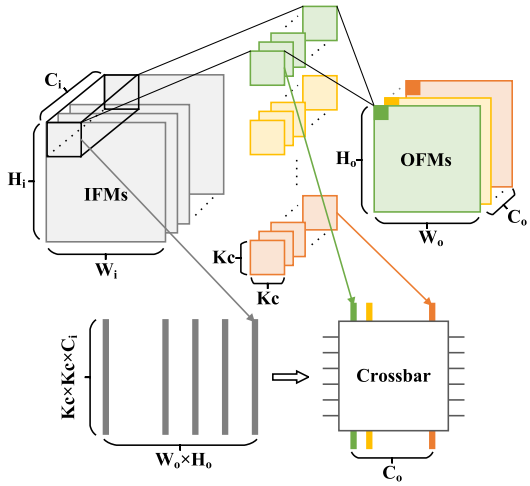


Fig. 2. Mapping convolution operations onto ReRAM-based crossbars.

— The experimental results reveal that the allocation scheme solved by our mathematical framework achieves 94% of the optimality. Besides, with the example of ISAAC [25], our framework optimizes the inference time of ReRAM-based CNN accelerators by 13.2×.
— By utilizing our framework, we analyze the relationship between the resource allocation scheme and the bandwidth constraints, to tell when the bandwidth starts limiting the performance. Our framework not only is an application-level optimizer, but also can provide suggestions on architectural design.

The rest of the paper is organized as follows. Section 2 introduces the background of this work. In Section 3 we present the proposed mathematical framework in detail. Section 4 presents and analyzes the experimental results. Finally, Section 5 concludes the paper.

2 BACKGROUND

2.1 Mapping CNNs onto ReRAM Crossbars

CNNs usually consist of Conv layers, **fully-connected (FC)** layers, pooling layers, **rectified linear unit (ReLU)** layers and so on. Conv and FC layers, due to their numerous dot-product computations, can be accelerated by ReRAM crossbars, while other layers are typically implemented by CMOS-based circuits. Figure 2 displays a general Conv layer in CNNs. It extracts characteristic elements from $C_i \times W_i \times H_i$[1] sized **input feature maps (IFMs)** with $C_o \times C_i \times K_c \times K_c$ sized weights and produces $C_o \times W_o \times H_o$ sized **output feature maps (OFMs)** as a result. Specifically, the output element of coordinate (x, y) on the z_{th} output feature map is calculated as Equation (1):

$$ofm[z][x][y] = \sum_{c=0}^{C_i} \sum_{k_x}^{K_c} \sum_{k_y}^{K_c} \left(\begin{array}{l} ifm[c][S_c x + k_x][S_c y + k_y] \\ \times weight[z][c][k_x][k_y] \end{array} \right) + bias[z], \quad (1)$$

where S_c denotes the stride size of the Conv layer. As for an FC layer, it is equivalent to a Conv layer with $W_i = H_i = K_c$ and $W_o = H_o = 1$. Hence, Conv and FC layers can naturally be modeled in a unified way.

[1]Those layer-related parameters, including C_i, C_o, W_i, W_o, H_i, H_o, K_c, and so on, are of a particular layer in a CNN. We ignore the layer index for these parameters if it will not lead to ambiguity. When necessary, layer indexes are marked as superscripts.

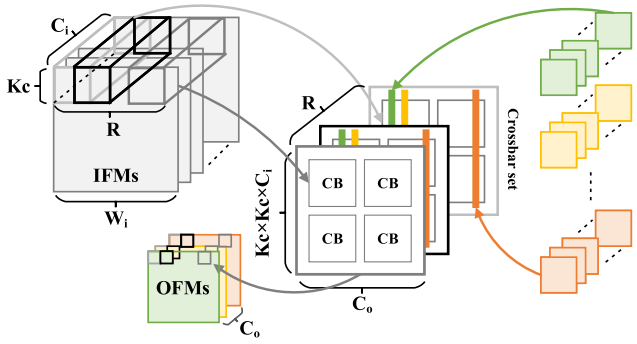


Fig. 3. Weight duplication.

ReRAM crossbars have widely been used for accelerating the MVM kernels in Conv and FC layers, based on the schematic explained in Figure 2. It illustrates the methodology for mapping convolution operations onto ReRAM crossbars. In the instance, the size of crossbars is $M \times N$. Weights from one filter are programmed into the same column of one or more crossbars, so a layer requires C_o columns in total, while inputs from one sliding window are applied to $K_c \times K_c \times C_i$ rows. It is important to emphasize that here crossbars refer to *logical crossbars*. A logical crossbar is the minimum resource to represent $M \times N$ weights under users' design specifications. A logical crossbar may include multiple physical $M \times N$ crossbars. As an example, if positive and negative weights use separate crossbars (like PRIME [7]), a logical crossbar has two physical crossbars. For another example, if the weights are 12 bits but a ReRAM cell can only store 4 bits, we need 3 physical crossbars to form a logical crossbar. If the two cases are both considered, the number of logical crossbars is $\frac{1}{6}$ of the number of physical crossbars. In this work, when we mention crossbars, we always refer to logical crossbars. But we use the size of a physical crossbar to represent the size of a logical crossbar. Considering the limited size of practical crossbars, it often needs multiple crossbars, denoted as a *crossbar set*, to map a single layer's weights. The number of crossbars in a crossbar set for a particular layer is

$$set = \left\lceil \frac{K_c \times K_c \times C_i}{M} \right\rceil \times \left\lceil \frac{C_o}{N} \right\rceil. \tag{2}$$

In fact, a crossbar set is the minimum resource to store a layer's weights. By using a crossbar set (set crossbars) for a layer, we can produce $1 \times 1 \times C_o$ outputs at a time. When the total number of crossbars in a ReRAM-based CNN acceleration system is more than $\sum_l set^l$ where the summation across all layers, if we still use the minimum resources to map the weights of all layers, there will be resource waste. Instead, a concept of *weight duplication* can be introduced, which maps the weights of a layer onto multiple crossbar sets. After duplicating the weights of a layer R times, as illustrated in Figure 3, R crossbar sets can compute in parallel with different inputs of that layer. As a result, intra-layer parallelism is obtained and performance is improved. $R \times C_o$ outputs can be produced in one step. It is emphasized that the number of crossbars for mapping a layer's weights is better to be a multiple of set (i.e., R is an integer) to simplify the hardware design. Partial sums generated by different crossbars in a crossbar set are accumulated by additional computational components in subsequent steps, if they belong to the same output element.

2.2 ReRAM-based CNN Accelerators

In recent years, ReRAM-based architectures have widely been used to accelerate CNNs (e.g., [1, 7, 8, 15, 22–25, 28–30, 33, 34]). Figure 4(a) shows a general abstract architecture of ReRAM-based

CNN accelerators, which is composed of multiple sub-chips (may be referred to tiles, process elements, clusters, etc. in different publications) connected with an inter-bus. Before processing CNN inference, weights after training are written into ReRAM crossbars as a one-time programming step. For each sub-chip, data from input buffers are computed in crossbars and the results are stored into output buffers. Figure 4(b) describes a typical execution flow for a Conv layer. First, the input data are converted to analog voltages through **digital-to-analog converters (DACs)**. Next, ReRAM crossbars complete the MVMs and the analog results are latched in sample-and-hold circuits. Finally, after the **analog-to-digital converters (ADCs)**, results are fed to the following shift-and-add circuits and ReLU/pooling components to get the final outputs, which are stored in output buffers through the on-chip bus. The number of cycles that each operation lasts depends on computing resources and/or the bandwidth. In the example of Figure 4(b), we assume to produce $1 \times 1 \times C_o$ outputs there needs four-cycle fetching inputs, one-cycle DAC and MVM, three-cycle ADCs and so on. Though different ReRAM-based CNN accelerators have different detailed implementations, from a high-level point of view, they generally follow a common abstract architecture and execution flow, which may be modeled in a unified way.

In conventional CMOS-based CNN accelerators, Conv layers are usually executed sequentially due to limited arithmetic resources. On the contrary, due to the plentiful computing resources in memory and also to avoid costly ReRAM re-programming, ReRAM-based accelerators typically enable all layers to be computed in parallel in a pipelined manner. Figure 4(e) explains how two adjacent Conv layers operate in a pipeline way, assuming $R = C_i = C_o = 1$ and $S_c = 1$ for both layers. As shown in Figure 4(e), ReRAM crossbars produce outputs row by row. After $(K_c^1 - 1) \times W_o^1 + K_c^1$ steps, Conv buffer 1 has accumulated enough outputs to activate the computation of Conv layer 2, as shown in step 1. Here accumulating enough outputs means that the accumulated outputs of a layer include all the inputs required by a sliding window of the next layer, so that the next layer can start the first convolutional computation. In step 2, Conv layer 1 generates a new output so that layer 2 can use it to compute the second result. That is to say, as long as the previous layer provides enough outputs, the current layer can use them to compute its outputs ("activated" by the previous layer), so that adjacent layers can be computed in such a pipelined way. In each step, the amount of inputs required by next layer is relevant to $R \times S_c$. This method realizes inter-layer parallelism. Combining with intra-layer parallelism brought by weight duplication, the two levels of parallelism together boost the performance of ReRAM-based CNN accelerators.

3 MATHEMATICAL FRAMEWORK FOR OPTIMIZING CROSSBAR ALLOCATION

As mentioned in the previous section, ReRAM-based CNN accelerators have both intra-layer and inter-layer parallelism. Weight duplication is a key factor that impacts the performance of both levels of parallelism. Duplicating the weights of a layer by more multiples may provide more intra-layer parallelism and boost the performance of that layer. Considering the inter-layer pipelined flow, the weight duplication multiples of all layers should be elaborated to achieve the best overall performance. Ideally, a ReRAM-based accelerator with abundant crossbars can complete the inference of a CNN in just L (L: number of layers in the CNN) steps with the help of maximum weight duplication, if other factors (e.g., bandwidth constraints) are ignored. Take VGG-19 [26] as an example, it needs 1387392 128×128 crossbars (about 23×10^9 ReRAM devices, assuming that a device can map a weight parameter) to ensure every layer is completed within one step. In practice, however, it is hard to provide such massive devices. Even with so many devices, it generates huge demands on memory access and the memory bandwidth will restrict the throughput. Thus, using maximum weight duplication such that every layer is completed in one step is impractical. How to allocate the given number of crossbars to layers needs to be discussed comprehensively. This is a fundamental problem but has not yet been well studied by previous works.

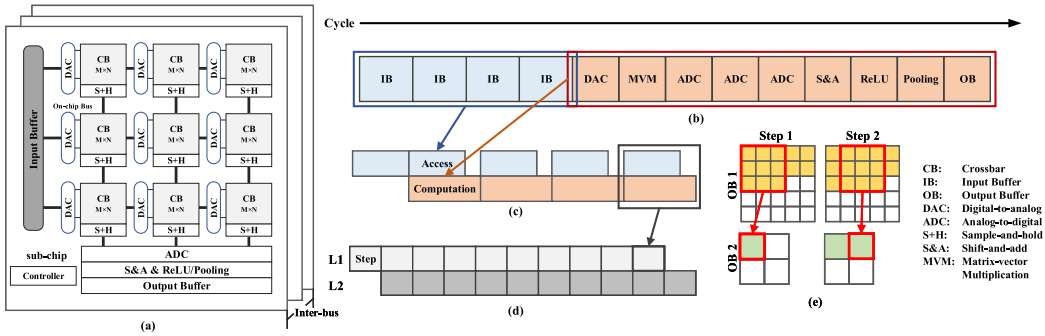


Fig. 4. (a) Abstract architecture of ReRAM-based CNN accelerators. (b) Typical execution flow of a Conv layer. (c) Two abstract pipeline stages of a layer. (d) Inter-layer parallelism. (e) Output buffers of two adjacent Conv layers.

In this section, we propose a mathematical framework to find the best crossbar allocation scheme, namely, the weight duplication multiples of all layers, to pursue the optimal performance of ReRAM-based accelerators with given number of crossbars and bandwidth constraints. This is a common problem of all ReRAM-based CNN accelerators. Before the introduction of our methodology, there are two points that need to be emphasized. First, our work is aimed at the situation where the number of crossbars is more than $\sum_{l=1}^{L} set^l$. Without this assumption, the CNN model is too large so that the number of crossbars is insufficient to store the weights of all layers, and weight re-writes are required in inference, which is impractical for ReRAM-based accelerators. Second, the optimization idea in our mathematical framework is based on a unified and abstract modeling for ReRAM-based CNN accelerators and can be applied to various architectural designs, and that will be explained below. Specifically, our proposed framework includes the following three steps.

(1) We define a unified parameter model to describe different layers and layer dimensions of CNNs.
(2) We propose a mathematical constrained optimization model that describes the crossbar allocation problem. We convert the practical limitations to mathematical constraints and quantify the relationship between performance and crossbar allocation, by modeling the essential computation and data access behaviors in the CNN inference flow in an abstract and unified way.
(3) We solve the optimization problem with dynamic programming. The obtained crossbar allocation scheme can minimize CNN inference time effectively.

3.1 Parameter Model

In order to describe different types of layers and different layer dimensions involved in practical CNNs, we construct a unified set of parameters to represent them. Three sets of parameters are included in the parameter model.

(1) **Architectural parameters:** We use $Total$ and $M \times N$ to represent the number and size of crossbars. Remember that $Total$ is the number of logical crossbars but not physical crossbars. In this work we use identical-size crossbars. This is reasonable because a practical memory system is typically composed of identical-size arrays.
(2) **CNN parameters:** For CNN parameters, the essential problem is how to describe different layers, such as pooling layers, Conv layers and FC layers, in a unified form. To solve this problem, we reorganize the original layers and describe the new layers with a

Table 1. Parameter Model of VGG-A

New layer	Original layer(s)	$\{C_i, C_o, W_o, H_o, K_c, K_p, S_c, S_p, P_c, P_p\}$
1	Conv-64, pooling	{3, 64, 112, 112, 3, 2, 1, 2, 1, 0}
2	Conv-128, pooling	{64, 128, 56, 56, 3, 2, 1, 2, 1, 0}
3	Conv-256	{128, 256, 56, 56, 3, 1, 1, 1, 1, 0}
4	Conv-256, pooling	{256, 256, 28, 28, 3, 2, 1, 2, 1, 0}
...	...	...
11	FC-4096 × 1000	{4096, 1000, 1, 1, 1, 1, 1, 1, 0, 0}

unified form $\{\boldsymbol{C_i}, \boldsymbol{C_o}, \boldsymbol{W_o}, \boldsymbol{H_o}, \boldsymbol{K_c}, \boldsymbol{K_p}, \boldsymbol{S_c}, \boldsymbol{S_p}, \boldsymbol{P_c}, \boldsymbol{P_p}\}$[2], where $\boldsymbol{K_c}/\boldsymbol{K_p}$, $\boldsymbol{S_c}/\boldsymbol{S_p}$, $\boldsymbol{P_c}/\boldsymbol{P_p}$ denote the kernel sizes of Conv/pooling layers, stride sizes of Conv/pooling layers, and padding sizes of Conv/pooling layers, respectively. As an instance, Table 1 lists the parameter model of the VGG-A model [26]. To use our CNN parameter model, some original layers are fused to form new layers. If $K_p = S_p = 1$, the new layer refers to a pure Conv layer. If $W_o = H_o = 1$, the new layer refers to an FC layer whose input/output size is C_i/C_o. In other cases, the new layer is a combination of a Conv layer and the following pooling layer. $\boldsymbol{W_o}$ and $\boldsymbol{H_o}$ always represent the output sizes of Conv layers. This unified representation of CNN parameters brings great convenience to the next steps, in which the constraints and objective are also unified.

(3) **Derived parameters:** In addition to the user-given parameters, we also use *set* to represent the number of crossbars in a crossbar set for a particular layer, which is defined in Equation (2).

3.2 Constrained Optimization Problem

We use $\boldsymbol{X}$ and $\boldsymbol{R}$ to represent the numbers of crossbars and crossbar sets allocated to the layers of a CNN, respectively. The crossbar allocation problem becomes finding the optimal $\boldsymbol{X}$ or $\boldsymbol{R}$ to minimize the inference time of an entire CNN. Based on the parameter model, we propose a constrained optimization model to generalize the problem. Section 3.2.1 defines the optimization problem. Section 3.2.2 models the CNN inference time in an abstract and unified way by considering the essential computations in CNN reference. Section 3.2.3 further complements the model by considering the memory bandwidth constraints.

3.2.1 Problem Definition. In our framework, we take the time required to complete a CNN's inference as the optimization objective. The constraints mainly come from the number of crossbars and the maximum weight duplication multiples, i.e.,

$$\sum_{l=1}^{L} X^l \le Total, \tag{3}$$

$$X^l = R^l \times set^l \text{ for } 1 \le l \le L, \tag{4}$$

$$1 \le R^l \le W_o^l \times H_o^l \text{ for } 1 \le l \le L,\ R^l \text{ is an integer}, \tag{5}$$

$$\text{architectural constraints on } \boldsymbol{R},\ \boldsymbol{X} \text{ or } \boldsymbol{set}. \tag{6}$$

Equations (3) and (4) are obvious. Due to Equation (4), finding the optimal $\boldsymbol{X}$ is equivalent to finding the optimal $\boldsymbol{R}$. For a layer l, the maximum weight duplication multiple is $W_o^l \times H_o^l$. If $R^l = W_o^l \times H_o^l$, the entire layer can be completed in one step. Equation (5) limits the maximum value

[2]A bold variable is a vector that is composed of the corresponding parameters of all layers.

of $\boldsymbol{R}$ to avoid unnecessary duplication. In addition, we limit $\boldsymbol{R}$ to be integers to simplify hardware design. If some R is not an integer, additional hardware components are needed to control the partial computation and data access which are related to the fractional part of R. For an integral R, the R-way parallelism is implemented in a unified way. For an FC layer ($W_o = H_o = 1$), it only requires an MVM operation, so $R = 1$ is already sufficient for its computation and it is the only possible duplication multiple for FC layers. In our framework, we offer an opportunity to set other constraints from the architectural perspective. For example, the ReRAM crossbars in one set prefer to be allocated in the same group [25], such as a bank or a tile, to minimize costly inter-group partial sum communication and structural harzards among different layers. In this case, the number of crossbars duplication in one group becomes an additional constraint for $\boldsymbol{R}$. Such constraints can be added in Equation (6). There are other restrictions like this, depending on the specific hardware design of ReRAM-based CNN accelerators.

3.2.2 Optimization Objective. The optimization objective is the CNN inference time, which should be accurately modeled. Figures 4(b)–4(e) illustrate the CNN inference flow from three different points of view. Assuming that a Conv layer i is allocated with R^i crossbar sets, each time it fetches $R^i \times K_c^i \times K_c^i \times C_i^i$ inputs and generates $R^i \times C_o^i$ outputs. We call this operation one *step*. Conv layer i needs $\lceil \frac{W_o^i \times H_o^i}{R^i} \rceil$ steps in total to complete its computation. Figure 4(b) illustrates the typical process of one step of a layer. Though different CNN accelerators have different detailed execution flows, basically, a step can always be divided into two abstract stages, a data access stage and a computation stage. Detailed latencies of them depend on the specific architectural design, dataflow and the assigned resources. The two stages are executed in a pipelined way, as explained in Figure 4(c), where any data access stage is for the next computation stage. In ReRAM-based CNN accelerators, all layers are computed in parallel. As long as in the previous step the previous layer has produced enough outputs, the current layer is activated and can be started, as Figure 4(d) shows. Based on the above modeling, it is obvious that CNN inference time can be calculated as

$$InferenceTime = T_{step} \times Op^L, \tag{7}$$

where T_{step} is the step latency and will be modeled in the next subsection. We use $\boldsymbol{Op}$ to represent the accumulated amounts of steps of all layers so that Op^L denotes the total number of steps of the entire CNN inference process. In this subsection, we first focus on modeling $\boldsymbol{Op}$ that is a function of $\boldsymbol{R}$, without considering the detailed latencies of data access and computation stages, which will be modeled in the next subsection.

Figure 5(a) illustrates a layer-wise pipelined inference process. ReRAM-based CNN accelerators process inference in a pipelined way as introduced in Section 2. Each layer can start its computation as long as the previous layer has produced sufficient outputs. Figure 5(a) points out that $\boldsymbol{Op}$ depends on the latencies of three kinds of operations, which is expressed as

$$\boldsymbol{Op} = \boldsymbol{NormalOp} + \boldsymbol{PreOp} + \boldsymbol{Nop}. \tag{8}$$

The computation of a layer can be described by the following process. First, to activate the computation of a layer, it requires a few steps waiting for sufficient inputs generated from the previous layer. The number of the preparation steps is called $PreOp$. After the preparation, under the ideal condition without any pipeline stall, the number of required steps for generating the output feature map of the layer is referred to as $NormalOp$. In each step, the layer can produce $R \times C_o$ outputs, so $NormalOp$ can be calculated by $\frac{W_o H_o}{R}$. However, due to pipeline imbalance, pipeline stalls may happen during the process. For example, when the computational throughput of the previous layer is much smaller than the current layer's, at a particular step, inputs required by the layer may not

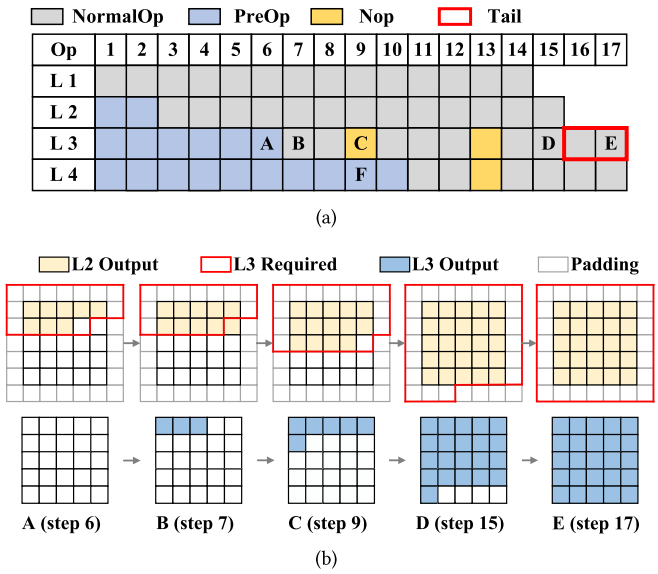


Fig. 5. (a) Pipelined timing diagram of CNN inference. (b) Output buffers of layers 2 and 3 in different steps. ($W_o^2 = 5, W_o^3 = 5, K_c^3 = 3, S_c^3 = 1, P^3 = 1, R^2 = 2, R^3 = 3, C_i^3 = 1, C_o^3 = 1$).

be available as the previous layer's computation is still in progress. In such cases, the computation of the current layer needs to temporarily pause. We call the number of the pauses as Nop.

We explain these operations in Figure 5(b) with the example of layers 2 and 3 in Figure 5(a). Layer 3 cannot start its computation until the 7th step due to the insufficient outputs of layer 2, so $PreOp^3 = 6$. Next, as $R^2 < R^3$, layer 2's outputs become insufficient in the 9th step, resulting in a nop operation in layer 3's execution. The same situation happens again in the 13th step. In the 15th step, layer 2 finishes its computation, but the existence of padding makes layer 3 continue its computation until the 17th step. The interval between the finish time of adjacent layers is called $Tail$ in our model.

Based on the inter-layer pipelined process, the CNN inference time is calculated using Algorithm 1. Conv layer i produces $R^i \times C_o^i$ outputs in every step, so its $NormalOp^i$ can be computed as line 17. From Figure 5(a) we can find that $PreOp^i$ consists of $PreOp^k$ and the time Conv layer k spends to activate Conv layer i ($k < i$), which is denoted as $\texttt{PreOpInterval}(i, k)$. Considering the popular Conv-pooling-Conv structure, we calculate $\texttt{PreOpInterval}(i, i-1)$ in lines 1-13. First, we calculate the number of rows and columns of outputs produced in Conv layer i's first operation, as shown in line 6. Next, taking (row, col) as inputs, we calculate the numbers of required rows and columns of pooling layer $i-1$ in lines 7-8. Similarly, to provide enough inputs for pooling layer i, $crow$ rows and $ccol$ columns are required in Conv layer $i-1$, as calculated in lines 9-10. Finally, $\texttt{PreOpInterval}(i, i-1)$ is calculated in line 12 with ($crow$, $ccol$). For the simpler Conv-Conv structure, this process is also applicable, as $K_p^{i-1} = S_p^{i-1} = 1$.

However, computing $PreOp^i$ with the above process may lead to a potential error. As shown in Figure 5(a), the operation marked at F for layer 4, which corresponds to $PreOp$, is actually also a nop operation caused by the nop operation at C. This additional stall cannot be calculated by $\texttt{PreOpInterval}(4, 3)$, which leads to an error in $PreOp^4$. However, we find that $\texttt{PreOpInterval}(4, 2)$ can take this stall into account and correct the result. This example explains why we traverse all the $i-1$ cases in line 18 for calculating the precise $PreOp^i$. For Nop^i, it is difficult to be modeled precisely, because it may appear in a variety of complex scenes and is hard

ALGORITHM 1: Calculation flow of CNN inference time.

Input: Number of layers L;
Parameters $\{C_i, C_o, W_o, H_o, K_c, K_p, S_c, S_p, P_c, P_p\}$;
Duplication of all layers R.
Output: CNN inference time.

```
// Calculate number of steps Conv layer k spends to activate Conv layer i
```

1 **Function** `PreOpInterval(`i, k`)`**:**
2 Initialize $Product \leftarrow R^i$;
`// ` W_p `denotes the width of pooling layers`
3 $W_p \leftarrow \frac{W_o - K_p + 2\times P_p}{S_p} + 1$;
4 **for** $j \leftarrow i$ **to** $k+1$ **do**
5 $row \leftarrow \left\lceil \frac{Product}{W_o^j} \right\rceil$;
6 $col \leftarrow Product - (row-1) \times W_o^j$;
7 $prow \leftarrow (row-1) \times S_c^j + K_c^j - P_c^j$;
8 $pcol \leftarrow \min\left\{(col-1) \times S_c^j + K_c^j - P_c^j, W_p^{j-1}\right\}$;
9 $crow \leftarrow K_p^{j-1} + S_p^{j-1} \times (prow-1) - P_p^{j-1}$;
10 $ccol \leftarrow \min\left\{K_p^{j-1} + S_p^{j-1} \times (pcol-1) - P_p^{j-1}, W_o^{j-1}\right\}$;
11 $Product \leftarrow \left\lceil \frac{(crow-1)\times W_o^{j-1} + ccol}{R^{j-1}} \right\rceil \times R^{j-1}$;
12 $pre \leftarrow \left\lceil \frac{(crow-1)\times W_o^k + ccol}{R^k} \right\rceil - 1$;
13 **return** pre ;

```
// Calculate CNN inference time
```

14 **Function** `InferenceTime(`R`)`**:**
15 Initialize $Op^1 = \left\lceil \frac{W_o^1 \times H_o^1}{R^1} \right\rceil$;
16 **for** $i \leftarrow 2$ **to** L **do**
17 $NormalOp^i \leftarrow \left\lceil \frac{W_o^i \times H_o^i}{R^i} \right\rceil$;
18 $PreOp^i \leftarrow \max_{0<k<i} \left\{\texttt{PreOpInterval}(i,k) + PreOp^k\right\}$;
19 $Tail^i \leftarrow \left\lceil \frac{W_o^i \times \lfloor P_c^i / S_c^i \rfloor}{R^i} \right\rceil$;
20 $Op^i \leftarrow \max\left\{NormalOp^i + PreOp^i,\ Op^{i-1} + Tail^i\right\}$;
21 $T_A^i \leftarrow \sum_{h \in hierarchy} \frac{DataSize_{i,h}}{BW_h}$;
22 $T_{step}^i \leftarrow \max\left\{T_A^i,\ T_C^i\right\}$;
23 $T_{step} \leftarrow \max_{1\le i\le L} \left\{T_{step}^i\right\}$;
24 **return** $Op^L \times T_{step}$;

to be analytically modeled. Instead, we introduce line 20 to estimate the total number of steps approximately. Moreover, line 20 involves $Tail$ to describe a special case in Figure 5(b). In this case, due to the existence of padding, layer 3 has to continue its computation even if there is no input produced from layer 2. We use $Tail$ to represent the overhead of these additional operations. Eventually, after traversing all layers in lines15-24, the total number of steps to complete the entire CNN inference is expressed as Op^L.

By minimizing Op^L under the constraints defined in Equations (3)–(6), one can get the optimal crossbar allocation scheme that minimizes the number of steps required to complete a CNN inference process. From Algorithm 1, we can draw the conclusion that the modeled number of computational steps to complete CNN inference only depends on the allocation scheme of ReRAM

crossbars, and has nothing to do with detailed architectural information. This is because we model the essential computations in the CNN inference process in an abstract and unified way. This also implies that our framework is applicable to different hardware architectures as long as they follow the general and abstract pipelined execution flow illustrated in Figure 4.

3.2.3 Bandwidth Consideration. Till now we have built the general relationship between ***Op*** and ***R*** without considering any architectural factors. People have pointed out that in CNN accelerators, communication (i.e., data access) which is constrained by the memory bandwidth is also an important factor that should be optimized [3, 9]. In order to take into account data access latency, T_{step} in Equation (7) should be modeled. In order to not only take into account the impact of architectural factors on data access latency, but also ensure the generality of our mathematical framework, in this subsection, we supplement the previous model with user's given architectural information and make a simplified but unified model for the data access latencies for all layers.

For each layer, data access and computation are executed in a pipelined way, as shown in Figure 4(c). The modeling of the latency of a computation stage (T_C^i) is trivial. In ReRAM-based CNN accelerators, computing resources are organized in basic processing units as the minimum granularity of computing resources. A processing unit consists of certain numbers of crossbars, DACs, ADCs and other components. As all the processing units in R^i crossbar sets compute in parallel, T_C^i only relates to the time taken for inputs to flow through a basic processing unit. It can include many factors, like the delays of ADCs and S&A units and so on. In ReRAM-based CNN accelerators, pooling layers are typically implemented by digital circuits and can hardly influence the inference time, so the delay can be included in T_C. Thus, T_C^i is a constant and has nothing to do with the crossbar allocation scheme.

If the memory bandwidth is sufficient, T_{step}^i equals to T_C^i and the CNN inference time is represented by Op^L. However, in fact, when the intra-layer parallelism ***R*** increases, the required amount of data in each clock cycle will also increase and the most important limiting factor will be the memory bandwidth, which will eventually become the performance bottleneck of the accelerator. Hence, it is interesting to figure out how the memory bandwidth affects the performance. With a limited memory bandwidth, computation and communication should be "balanced" in some way. Without loss of generality, we consider an architecture with multiple memory hierarchies, so the data access latency T_A^i will be accumulated by the time of fetching inputs from the farthest memory hierarchy to the nearest memory hierarchy.

Based on the above analysis, the bandwidth-related parameters are modeled as

$$T_A^i = \sum_{h \in hierarchy} \frac{DataSize_{i,h}}{BW_h}, \tag{9}$$

$$T_{step}^i = \max\left\{T_A^i,\ T_C^i\right\}, \tag{10}$$

$$T_{step} = \max_{1 \le i \le L}\left\{T_{step}^i\right\}, \tag{11}$$

where $DataSize_{i,h}$ means the size of data that are located in memory hierarchy h and need to be transferred to layer i, and BW_h denotes the corresponding bandwidth. It is emphasized that though the calculation of Equation (9) depends on architectural information, the model is universal as Equation (9) only involves very few common parameters of any architectures. We take ISAAC [25] as an example to explain how to use Equation (9). In ISAAC, as Figure 4(a) displays, each tile (corresponding to a sub-chip in Figure 4(a)) has an input buffer and an output buffer. Data within a tile flow through an on-chip bus, while data from different tiles are communicated with an inter-bus. According to the architectural information of ISAAC, we find that generally one CNN layer

will be mapped to multiple tiles. Denote the number of crossbar sets in one tile as r, the number of tiles in layer i and layer $i-1$ as T^i and T^{i-1}, respectively. The intra-tile data access size equals to $r^i \times K_C^i \times K_C^i \times C_i^i$. When layer i needs to access inputs from layer $i-1$, all T^{i-1} tiles' results should be broadcast to T^i tiles, which means that the data size communicated across tiles is $T^i \times T^{i-1} \times r^{i-1} \times C_o^{i-1}$. Considering the worse case, T_A^i is the sum of both intra-tile and inter-tile data access latencies, i.e.,

$$T_A^i = \frac{r^i \times K_C^i \times K_C^i \times C_i^i}{BW_{intra-tile}} + \frac{T^i \times T^{i-1} \times r^{i-1} \times C_o^{i-1}}{BW_{inter-tile}}. \tag{12}$$

In Section 3.2.2, we have modeled $\boldsymbol{Op}$ (number of steps to complete all layers) as a function of $\boldsymbol{R}$. Once given the abstract architectural information, the proposed mathematical framework can be used for various architectures by taking into account detailed latencies of computation and data access. As shown in lines 21-23 of Algorithm 1, with the given crossbar allocation scheme and bandwidth information, we can calculate T_{step}, which in turn, decides the inference time of the CNN model in line 24. Since both $\boldsymbol{Op}$ and T_{step} depend on the crossbar allocation scheme $\boldsymbol{R}$, how to find the optimal crossbar allocation scheme to balance communication and computation, as well as to minimize the inference time of the entire CNN model, is also a key problem. In the next subsection we propose a solver for the constructed optimization problem.

3.3 Dynamic Programming based Solver

In the previous subsection, we have constructed a constrained optimization problem to optimize crossbar allocation for ReRAM-based CNN accelerators. The size of the search space is the number of positive integer solutions $\{R^1, \ldots, R^L\}$ of the inequality

$$set^1 R^1 + set^2 R^2 \cdots + set^L R^L \leq Total, \tag{13}$$

which is a linear Diophantine inequality. As far as we know, there is no closed form expressing the number of positive integer solutions of a general linear Diophantine inequality. Considering a simpler case where all set^l $(l = 1, 2, \ldots, L)$ values are identical (denoted as set), the number of positive integer solutions of $R^1 + R^2 + \cdots + R^L = Total/set = N$ is simply C_{N-1}^{L-1}, where C_n^k is the number of selections of k items from n items. This can be explained as follows. $L-1$ separators can be placed among N horizontally placed identical items to partition the N items into L non-empty subsets (the number of items in the l-th subset is R^l), where a partitioning one-to-one corresponds to a positive integer solution of the simpler case. Clearly, there are $N-1$ positions in which each can hold one separator at most, so the number of selections is C_{N-1}^{L-1}. Accordingly, for Equation (13), the number of positive integer solutions is roughly $O(C_{Total/\ \overline{set}-1}^{L-1})$ where $\overline{set}$ is the mean value of $\boldsymbol{set}$. The actual complexity is larger because the use of $\overline{set}$ to derive the complexity simplifies the problem and reduces the complexity. It is typically an astronomical search space that is impossible to be fully traversed. Based on our analysis, we can find that the problem has the optimal substructure property of dynamic programming, which means that the minimum inference time cost by layers 1 to l is determined by the larger one between the minimum time cost by layers 1 to $l-1$ and the inference time of layer l, so that the optimal solution of the problem should obey the property shown in lines 9–13. Hence, we employ dynamic programming to quickly find the near-optimal crossbar allocation scheme, as described in Algorithm 2.

Algorithm 2 aims at optimally allocating $Total$ crossbars across L layers to maximize the accelerator performance. In Algorithm 2, we construct two 2D matrices, $\boldsymbol{R_{opt}}$ and $\boldsymbol{T_{min}}$, which respectively record the optimal crossbar allocations and the minimal inference time for different problems. Each problem is specified by variables i and G, where i varies from 1 to L, and G ranges

ALGORITHM 2: Dynamic programming based solver.

Input: Number of layers L;
Parameters $\{\boldsymbol{C_i}, \boldsymbol{C_o}, \boldsymbol{W_o}, \boldsymbol{H_o}, \boldsymbol{K_c}, \boldsymbol{K_p}, \boldsymbol{S_c}, \boldsymbol{S_p}, \boldsymbol{P_c}, \boldsymbol{P_p}\}$;
Number of crossbars in crossbar sets $\boldsymbol{set}$.
Output: The optimal allocation scheme $\boldsymbol{R_{opt}}[L, Total]$.

1 Initialize $\boldsymbol{T_{min}} \leftarrow +\infty$;
2 **for** $i \leftarrow 1$ **to** L **do**
3 **for** $G \leftarrow \sum_{k=0}^{i} set^k$ **to** $Total - \sum_{k=i+1}^{L} set^k$ **do**
4 **if** $i = 1$ *and* $(G \bmod set^1) = 0$ **then**
5 $\boldsymbol{R_{opt}}[1, G] \leftarrow \lceil \frac{G}{set^1} \rceil$;
6 $T_{min}[1, G] \leftarrow$ `InferenceTime` $\boldsymbol{R_{opt}}[1, G]$;
7 **else**
8 **for** j *that satisfies the constraints in Equations* (3)–(6) **do**
9 $\boldsymbol{R_{temp}} \leftarrow \boldsymbol{R_{opt}}[i-1, G-j] + \lceil \frac{j}{set^i} \rceil$;
10 $T_{temp} \leftarrow$ `InferenceTime`$(\boldsymbol{R_{temp}})$;
11 **if** $T_{temp} < T_{min}$ **then**
12 $\boldsymbol{R_{opt}}[i, G] \leftarrow \boldsymbol{R_{temp}}$;
13 $T_{min}[i, G] \leftarrow T_{temp}$;

between $\sum_{k=0}^{i} set^k$ and $Total - \sum_{k=i+1}^{L} set^k$, as depicted in lines 2-3. Specifically, $\boldsymbol{R_{opt}}[i, G]$ indicates the optimal solution of allocating G crossbars to layers 1 to i, and $T_{min}[i, G]$ is the CNN inference time of the corresponding problem, which is calculated using Algorithm 1. The optimal crossbar allocation is finally determined by $\boldsymbol{R_{opt}}[L, Total]$, which can be derived through iteratively solve its sub-problems, where $i \leq L$ and $G \leq Total$. Lines 4–6 imply that if there is only one layer, the optimal solution is exactly allocating all crossbars to it. Subsequently, in lines 8-13, $\boldsymbol{R_{opt}}[i, G]$ is iteratively solved through traversing its sub-problems $\boldsymbol{R_{opt}}[i-1, G-j]$ over j, where j represents the number of crossbars allocated to layer i. In each iteration, j and $\boldsymbol{R_{opt}}[i-1, G-j]$ are combined for generating an intermediate variable R_{temp}. $\boldsymbol{R_{opt}}[i, G]$ represents the corresponding R_{temp} when its inference time is minimized. The dynamic programming based solver is expected to find the optimal solution. However, as analyzed in Section 3.2.2, the modeling of ***Nop*** is not completely precise so the solution may not be globally optimal. We will show by experiments in the next section that the solved solutions are almost identical to the results of exhaustive/random searches, with negligible differences, while our solver is much faster than exhaustive searches.

It can easily be derived that the time complexity of Algorithm 2 is $O(L^2 \times Total^2)$ (the time complexities of the loops at lines 2, 3 and 8 are $O(L)$, $O(Total)$ and $O(Total)$, respectively, and the time complexity of `InferenceTime` is $O(L)$), which is much lower than that of exhaustive searches.

4 EXPERIMENTAL RESULTS

Benchmarks: To verify the correctness and effectiveness of the proposed mathematical framework, we benchmark five popular CNN models, including AlexNet [18], VGG-A [26], VGG-E [26], ResNet-18 [13] and MobileNet-v1 [14]. Table 2 lists the parameter models of these CNNs, where each grid contains a specific parameter list of all layers in a CNN model. It should be noted that a layer here actually refers to a fused layer, instead of an original layer, according to our parameter model proposed in Section 3.1.

Evaluation Methodology and Baselines: To obtain the ground-truth of the number of steps needed for the inference of CNN models, we design a behavior-level simulator, which simulates the CNN inference process at the cycle level. The simulator has been verified against an

Table 2. Parameter Models of Benchmarks

Para.	AlexNet	VGG-A	VGG-E	ResNet-18
C_i	[3, 96, 256, 384, 384]	[3, 64, 128, 256#2[a], 512#3]	[3, 64#2, 128#2, 256#4, 512#7]	[3, 64#5, 128#4, 256#4, 512#3]
C_o	[96, 256, 384, 384, 256]	[64, 128, 256#2, 512#4]	[64#2, 128#2, 256#4, 512#8]	[64#5, 128#4, 256#4, 512#4]
W_o	[55, 27, 13, 13, 13]	[224, 112, 56#2, 28#2, 14#2]	[224#2, 112#2, 56#4, 28#4, 14#4]	[112, 56#4, 28#4, 14#4, 7#4]
K_c	[11, 5, 3, 3, 3]	[3, 3, 3, 3, 3, 3, 3, 3]	[3#16]	[7, 3#16]
K_p	[3, 3, 1, 1, 3]	[2, 2, 1, 2, 1, 2, 1, 2]	[1, 2, 1, 2, 1#3, 2, 1#3, 2, 1#3, 2]	[3, 1#16]
S_c	[4, 1, 1, 1, 1]	[1, 1, 1, 1, 1, 1, 1, 1]	[1#16]	[2, 1#4, 2, 1#3, 2, 1#3, 2, 1#3]
S_p	[2, 2, 1, 1, 2]	[2, 2, 1, 2, 1, 2, 1, 2]	[1, 2, 1, 2, 1#3, 2, 1#3, 2, 1#3, 2]	[2, 1#16]
P_c	[3, 3, 1, 1, 3]	[1, 1, 1, 1, 1, 1, 1, 1]	[1#16]	[3, 1#16]
P_p	[0, 0, 0, 0, 0]	[0, 0, 0, 0, 0, 0, 0, 0]	[0#16]	[1, 0#16]

[a]The notation $parameter\#repeat$ means that $parameter$ is repeated $repeat$ times (for simplicity and to save space).

Table 3. Parameters of ISAAC-like Architecture [25]

Component	Parameter	Value
Tiles (sub-chips in Figure 4(a))	Number	Variable[b]
Crossbars per tile	Size, number	128×128, 72
ADCs per crossbar	Resolution, number	8-bit, 1
DACs per crossbar	Resolution, number	1-bit, 128
On-chip buffer	Bandwidth	128 GB/s
Inter-bus	Bandwidth	12.8 GB/s
Computation stage	Latency	21 cycles
Clock	Period	100ns
Inputs/weights	Precision	16-bit

[b]Since we will compare our method with ISAAC with different number of crossbars, the number of tiles is set to a variable.

open-source simulator, MNSIM [36], using the built-in CNN models (LeNet, AlexNet, VGG8, VGG16 and ResNet18) of the MNSIM package. Since MNSIM does not support weight duplication, our simulations without weight duplication show that the difference between the results of our simulator and MNSIM is 9.92% on average. The simulation results are used to evaluate the accuracy of the inference time modeling. To validate the optimality of the solutions obtained by the proposed crossbar allocation methodology, we compare our method with exhaustive/random searches. Our method is also compared with three heuristics. Each of them corresponds to a crossbar allocation strategy proposed in a previous work. Specifically, PipeLayer [28] ensures the crossbar duplication ratios proportional to $\boldsymbol{W_o} \times \boldsymbol{H_o}$[3]. ISAAC [25] applies the heuristic S_c^2-based weight duplication and PRIME [7] adopts a method similar to the identical weight duplication which we will evaluate. We further make detailed comparisons with an ISAAC-like architecture whose key architectural parameters are summarized in Table 3. The ISAAC-like architecture has the same structure as ISAAC, but with different parameters, like the number of tiles and the number of crossbars in one tile.

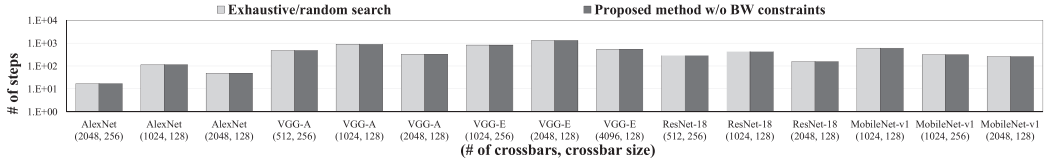
4.1 Accuracy of Inference Time Modeling

As mentioned in Section 3.2, since $\boldsymbol{Nop}$ is difficult to be precisely modeled, the inference time evaluated by Algorithm 1 may not be precise, either. Table 4 lists the **error rate (ER)** of the evaluated number of steps needed for CNN inference, taking the results of behavior-level

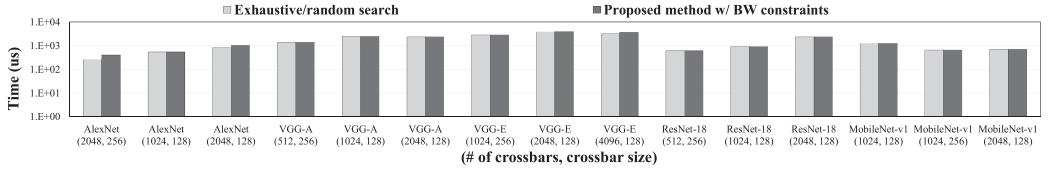
[3]In this paper, a multiplication between two vectors means an element-wise multiplication and the result is still a vector.

Table 4. Inference Time Modeling Error and Accuracy

CNN model	ER <1%	ER 1 − 5%	ER >5%	Average accuracy
AlexNet	89.2%	8.4%	2.4%	99.6%
VGG-A	64.8%	34.7%	0.5%	99.1%
VGG-E	51.4%	47.7%	0.9%	98.8%
ResNet-18	67.7%	30.1%	2.2%	98.9%
MobileNet-v1	60.2%	34.1%	5.7%	97.1%



(a)



(b)

Fig. 6. Inference time of solutions obtained by our framework and exhaustive/random searches. (a) Without bandwidth constraints. (b) With bandwidth constraints.

simulation as the ground-truth. We simulate the CNN inference process cycle by cycle, and make a comparison with the calculated Op^L. We randomly generate 10,000 crossbar allocation schemes and collect the proportions of cases with ≤1% error, 1-5% error, and ≥5% error, respectively. As Table 4 shows, for most cases, the error rate is below 5%. The maximum value of ER is 15%. For cases with slightly large ERs, they are caused by large $\boldsymbol{Nop}$ values as $\boldsymbol{Nop}$ is the only error source in our inference time model. Large $\boldsymbol{Nop}$ values generally imply long inference time. Solutions of long inference time will be eliminated by our solver. Hence, the cases with slightly large ERs can hardly impact the solution optimality of our solver. The average modeling accuracy is higher than 98%, indicating the correctness of our inference time model.

One may ask a question that since we have a behavior-level simulator, why we use the proposed inference time model, which is not 100% precise, instead of the simulator, for measuring the inference time in the dynamic programming based solver. In fact, we have compared the runtime between them. In order to simulate one case, the simulator takes about 1000× longer time than our model. If we use the simulator in our solver, it will take a few months to complete the largest case we have tested. Therefore, it is impractical to adopt the simulator in our solver.

4.2 Solution Optimality

Our proposed mathematical framework is aimed at finding the near-optimal crossbar allocation scheme for minimizing CNN inference time, regardless of the accelerator architecture details. To prove that, we compare the solutions solved by our framework with the optimal results obtained by exhaustive searches or random searches with pruning, as shown in Figure 6. With pruning, the search space shrinks a lot by eliminating many intuitively bad solutions. To determine whether a

crossbar allocation candidate solution can be pruned, we calculate the cosine similarity between $\boldsymbol{R}$ and $\boldsymbol{W_o} \times \boldsymbol{H_o}$. Solutions with small cosine similarity means that the crossbar allocation schemes and the workloads of layers significantly mismatch, so that they tend not to be the optimal solution. Each benchmark is tested with several cases which are denoted as (# of crossbars, crossbar size[4]). The tested cases cover different situations. For example, AlexNet (2048, 256) represents the case where abundant crossbars are allocated for small-scale CNNs, while VGG-E (2048, 128) represents the case with a small number of crossbars allocated to large-scale CNNs. To obtain the optimal results, for small-scale cases, exhaustive searches are performed; while for large-scale cases, due to the extremely huge search space, we have to run a number of (10^8) random searches after pruning to find a quasi-optimal solution.

The results shown in Figure 6(a) are without the bandwidth constraints so they describe the number of steps for CNN inference. The figure intuitively reveals that there is almost no difference in the inference time between the solved solutions and the exhaustive/random search (pruning applied) results, which implies the near-optimality of the proposed framework. There are three cases for which the inference time obtained by our method is worse than that obtained by exhaustive/random searches with pruning, AlexNet (1024, 128), VGG-A (1024, 128) and VGG-E (2048, 128). They spend 0.43% more steps than optimum/quasi-optimum on average. The small error is caused by the approximate modeling of $\boldsymbol{Nop}$, as mentioned in Section 3.2.2.

When considering bandwidth constraints, Figure 6(b) compares the inference time obtained by the proposed method with the optimal/quasi-optimal solutions obtained by exhaustive/random searches with pruning. Our solutions achieve 94% of the optimal/quasi-optimal performance on average, which implies the bandwidth consideration may affect the optimality of our framework a little. However, we will show in the following comparisons that our method is still far better than heuristics and ISAAC [25].

4.3 Comparison with Heuristics

In Figure 7, we compare our crossbar allocation strategy with three heuristics (without bandwidth constraints): $\boldsymbol{W_o H_o}$-proportional duplication, $\boldsymbol{S_c^2}$-based duplication and identical duplication. The three heuristics are explained as follows.

In the first heuristic, $\boldsymbol{R}$ is proportional to $\boldsymbol{W_o} \times \boldsymbol{H_o}$. As a crossbar set stores exactly one copy of the weights of a layer and can implement $K_c \times K_c \times C_i \times C_o$ MACs at a time, $\boldsymbol{W_o H_o}$-proportional duplication makes the number of MACs that the crossbars allocated to a layer can complete at a time be proportional to $K_c \times K_c \times C_i \times C_o \times W_o \times H_o$, which is the total workload of a layer. This implies that $\boldsymbol{W_o H_o}$-proportional duplication allocates crossbars to layers based on the workloads of layers and tends to balance the number of steps among all layers. This is a straightforward approach that has obvious intuitive insight. PipeLayer [28] uses this heuristic to balance the stage latencies of the pipeline.

The second heuristic, $\boldsymbol{S_c^2}$-based duplication, has been adopted in ISAAC [25]. It claims that if the weight duplication multiple of layer i is R^i, R^{i-1} should be $(S_c^i)^2 \times R^i$ (S_c is the convolution stride size). The intuitive meaning is to balance the computation stages of the inter-Conv pipeline. Based on this principle, the weight duplication multiples of all layers are determined in the layer reversed order, from the last layer to the first layer. The stride sizes of pooling layers are not considered in the calculation of the duplication multiples.

In the last heuristic, $\boldsymbol{R}$'s elements are identical, meaning that all layers' weights are duplicated by the same multiple. This means that all layers are allocated the same number of crossbar sets. Note that the numbers of crossbars allocated to all layers are not necessarily equal, since the number of

[4]Square crossbars (i.e., $M = N$) are used in all experiments.

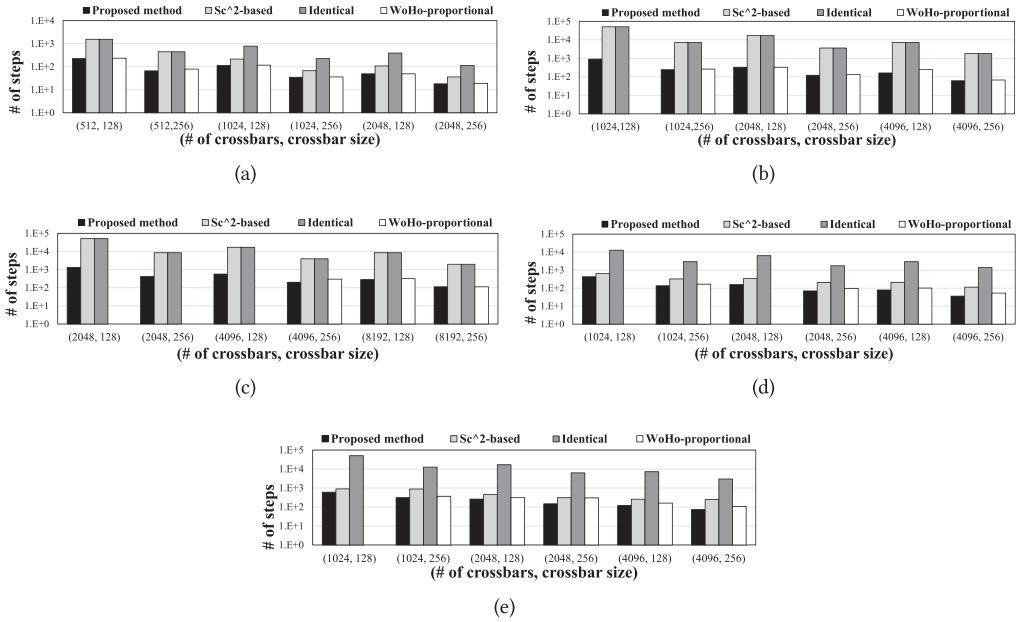


Fig. 7. Inference time (number of steps) comparisons with three heuristics, without bandwidth constraints. (a) AlexNet. (b) VGG-A. (c) VGG-E. (d) ResNet-18. (e) MobileNet-v1.

crossbars in a crossbar set is different for different layers, as expressed in Equation (2). S_c^2-based duplication and identical duplication generate identical results for VGG-A and VGG-E, as their $S_c = 1$.

From Figure 7, we can find that our framework generates better results than the three heuristics. The average improvements in the inference time (number of steps) against $W_o H_o$-proportional duplication, S_c^2-based duplication and identical duplication are 1.18×, 15.1× and 32.03×, respectively. Though $W_o H_o$-proportional duplication generates somewhat similar results as our method, it ignores many practical factors and has several shortcomings.

In some cases, there are no data in Figure 7 for the $W_o H_o$-proportional duplication method, because the given number of crossbars is not sufficient for carrying out that crossbar allocation strategy. In fact, this is one of the major drawbacks of $W_o H_o$-proportional duplication. Since $W_o \times H_o$ usually vary much for all layers in a CNN, even when the layer with the smallest $W_o \times H_o$ has the smallest weight duplication multiple, namely, 1, the layer with the largest $W_o \times H_o$ may still need a large number of crossbars, making the total crossbar requirement exceed the given crossbar number limit.

In practice, the given number of crossbars may not be fully allocated to layers by the heuristics, and there may be some remaining crossbars that are insufficient for another duplication of weights for all layers. Table 5 analyzes such a situation. It compares the inference time (number of steps) between our method and $W_o H_o$-proportional duplication for five cases. The proposed framework can always fully utilize the given crossbars, while the $W_o H_o$-proportional duplication heuristic leads to some remaining crossbars. As a result, our method improves the inference time. For the five test cases, the average inference time improvement is 28.3%.

Most important of all, the heuristics cannot handle bandwidth constraints but our framework can. Practical hardware architectures always have bandwidth constraints. The results with bandwidth constraints taken into account will be presented in the next subsection.

Table 5. Inference Time (Number of Steps) Comparison and Number of Remaining Crossbars of $W_o H_o$-Proportional Duplication Method

Case	Proposed method # of steps	$W_o H_o$-proportional # of steps	$W_o H_o$-proportional # of remainders
VGG-A (4096, 128)	162	245	253
VGG-E (8192, 128)	280	318	514
VGG-E (4096, 256)	201	295	100
ResNet-18 (4096, 128)	79	101	278
MobileNet-v1 (4096, 128)	147	303	106

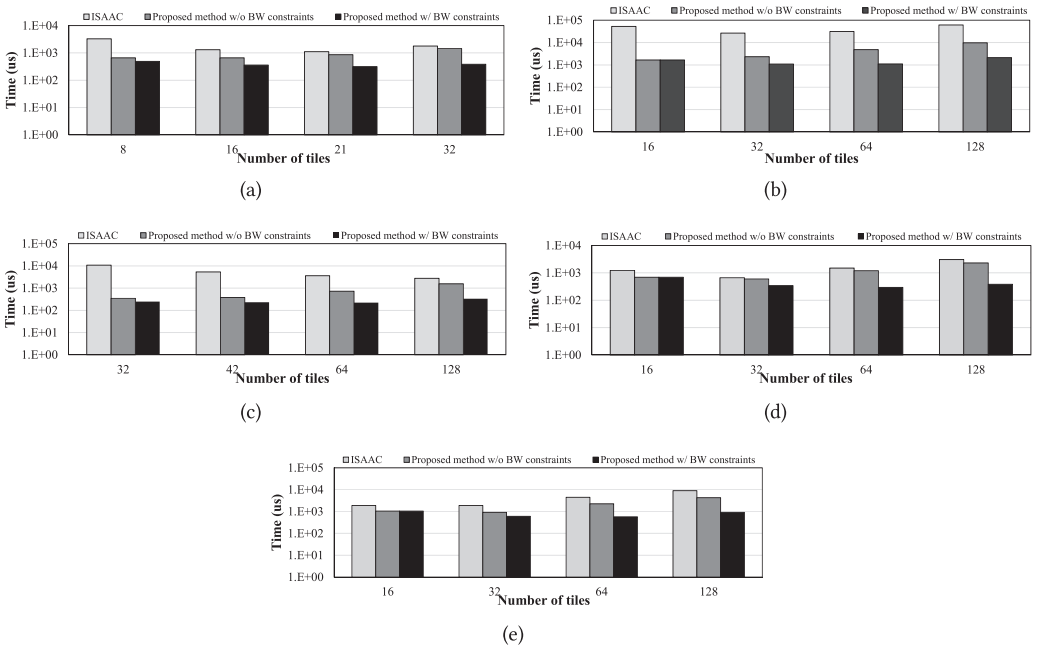


Fig. 8. Inference time comparison with ISAAC. (a) AlexNet. (b) VGG-A. (c) VGG-E. (d) ResNet-18. (e) MobileNet-v1.

4.4 Comparison with ISAAC with Bandwidth Constraints

To validate the effectiveness of our mathematical framework when applied to existing ReRAM-based CNN accelerators, we compare the inference time between our method (with and without bandwidth constraints) and ISAAC (with bandwidth constraints), as shown in Figure 8. We use the architectural parameters listed in Table 3 to evaluate our method and ISAAC, while the number of tiles (corresponding to sub-chips in the abstract architecture illustrated in Figure 4(a)) varies. Each tile has 72 128×128 crossbars.

Compared with the S_c^2-based duplication method adopted by ISAAC, our method improves the inference time significantly. The average improvements in the inference time against ISAAC are 6.6× and 13.2×, respectively, for our method without and with bandwidth constraints. We can find that ISAAC's allocation strategy that tries to balance the computation stages of the inter-Conv pipeline introduces unbalanced data access latencies and Conv-pooling pipeline structure, leading to an apparent performance decrease. With the increase of the crossbar number, the inference

Table 6. Analysis on Optimal Crossbar Allocation

Case		Optimal allocation scheme & features								
		L1	L2	L3	L4	L5	L6	L7	L8	σ^c
AlexNet(2304, 128)	R	106	21	7	6	6				
	$W_o \times H_o$	3025	729	169	169	169				
	$W_o \times H_o/R$	28.54	34.71	24.14	28.17	28.17				3.53
	T_{step}	2.10	31.17	5.58	2.42	2.11				12.66
VGG-A(2304, 128)	R	200	50	13	13	4	4	1	1	
	$W_o \times H_o$	50176	12544	3136	3136	784	784	196	196	
	$W_o \times H_o/R$	250.88	250.88	241.23	241.23	196	196	196	196	27.00
	T_{step}	2.10	7.57	3.86	3.52	2.10	2.10	2.10	2.10	1.92
AlexNet(2304, 128)	R	26	6	2	22	2				
	$W_o \times H_o$	3025	729	169	169	169				
	$W_o \times H_o/R$	116.35	121.50	84.50	7.68	84.50				52.25
	T_{step}	2.10	2.25	2.10	2.54	2.56				0.23
VGG-A(2304, 128)	R	112	28	10	10	5	4	2	2	
	$W_o \times H_o$	50176	12544	3136	3136	784	784	196	196	
	$W_o \times H_o/R$	448.00	448.00	313.60	313.60	156.80	196	98	98	143.26
	T_{step}	2.10	2.20	2.10	2.10	2.10	2.10	2.10	2.10	0.04

[c]Standard deviation of the corresponding row.

time of the solutions found by our method without bandwidth constraints decreases continuously, which implies that data access cost gradually dominates the overall performance. In this case, optimization without considering bandwidth constraints may not work well. This comparison illustrates the necessity of considering the memory bandwidth in the optimization problem. Nevertheless, our method without bandwidth constraints with pooling stride taken into account can still achieve better performance compared with ISAAC's, thanks to the unified layer description model. Besides, our method with bandwidth constraints can further balance both computation and communication well, so the obtained inference performance is superior and stable.

4.5 Result Analysis

We have shown by comprehensive results that our framework is able to find near-optimal crossbar allocation schemes under the given number of crossbars. Furthermore, we would like to find out some intuitive explanations behind the found solutions, which also helps generalize the principles for allocating crossbars for ReRAM-based CNN accelerators, as listed in Table 6. Case 1 is AlexNet (2304, 128) and case 2 is VGG-A (2304, 128), without bandwidth constraints. Cases 3 and 4 are same as cases 1 and 2, respectively, with bandwidth constraints taken into account. Cases 3 and 4 use the same architectural parameters as in Figure 8, which are listed in Table 3, where 2,304 crossbars correspond to 32 tiles.

Without considering bandwidth constraints, we find that the optimal duplication multiples $\boldsymbol{R}$ are approximately proportional to $\boldsymbol{W_o \times H_o}$ (for cases 1 and 2, $W_o \times H_o/R$ is similar for all layers with a small standard variation) to minimize the inference time. In this case, the workloads of all layers can be well balanced and nop operations are also minimized. This is a universal conclusion generalized from the bandwidth-unconstrained optimal crossbar allocation schemes, which also explains why $\boldsymbol{W_oH_o}$-proportional duplication can generate slightly worse results than our method, as shown in Figure 7. However, from the solutions obtained by our framework, the optimal duplication multiples are not strictly proportional to $\boldsymbol{W_o \times H_o}$. The reasons are mainly caused by the practical limitations of the $\boldsymbol{W_oH_o}$-proportional duplication method, which have been explained in the last few paragraphs of Section 4.3.

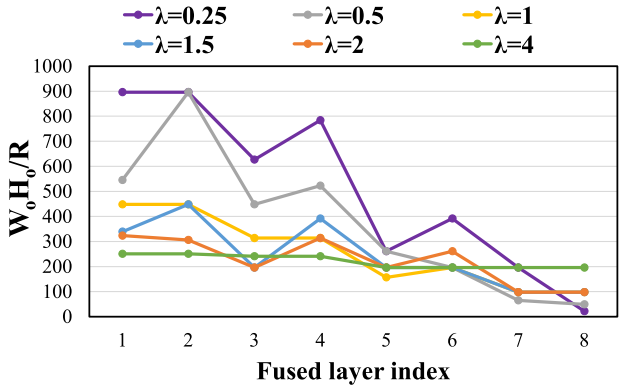


Fig. 9. $W_o \times H_o/R$ ratios under different bandwidth constraints (layers are fused by our parameter model presented in Section 3.1 so the X-axis shows the indexes of the fused layers instead of the original layers).

When memory bandwidth becomes the bottleneck which limits the data access performance, the above conclusion no longer holds. In this case, our method with bandwidth constraints tends to generate balanced computation and communication, and balanced bandwidth utilization for all layers, as suggested by the small standard deviation of T_{step} in cases 3 and 4 of Table 6. From this analysis we can conclude that **when the data access overhead becomes the bottleneck, the optimal crossbar allocation solution cannot be easily derived from universal heuristic rules and will vary depending on the CNN model and hardware resources**, which further explains the necessity of using our mathematical framework by taking into account memory bandwidth constraints for solving practical problems.

We further analyze the relationship between the bandwidth constraints and the optimal crossbar allocation scheme, to see when the bandwidth starts limiting the performance and influencing the crossbar allocation results. For this purpose, we scale the bandwidth values listed in Table 3 by a factor λ. We use VGG-A (2304, 128) (cases 2 and 4 in Table 6) as an example to illustrate this relationship. The results are shown in Figure 9. If the bandwidth scale factor is 4, the $W_o \times H_o/R$ ratios for all layers are almost identical, indicating that the bandwidth is sufficiently large and is not the performance bottleneck. When the bandwidth scale factor is 2, the $W_o \times H_o/R$ ratios are slightly different. When the bandwidth scale factor becomes smaller than 2, in other words, when the on-chip buffer and inter-bus bandwidths are smaller than 256 GB/s and 25.6 GB/s, respectively, the $W_o \times H_o/R$ ratios for all layers start becoming different. From this point, the bandwidth starts becoming the performance bottleneck. It also implies that in this case the bandwidth that the hardware architecture can provide is insufficient. From this analysis, one may provide suggestions on the bandwidth requirements for a given architecture. By doing so, the significance of this work is extended beyond its original usage of optimizing crossbar allocation for CNNs.

4.6 Runtime of Proposed Solver

Our dynamic programming based solver implemented in Python is fast. Table 7 lists the runtime of our solver for some cases. For exhaustive/random searches with pruning, we remove some intuitively impossible cases to shorten the searching time, but it still takes a very long time to get results. For small-scale cases, the runtime of our solver is at the magnitude of seconds to minutes, while exhaustive searches need several minutes to days. For the largest case we have tested, our solver needs about 4 hours to get the solution. For the same case, if we traverse the entire search space, the exhaustive search time will be more than 10^{10} years, estimated based on the time of 10,000

Table 7. Runtime Results of Dynamic Programming based Solver

Case	Time	Case	Time
AlexNet(2048, 128)	11 sec.	AlexNet(4096, 256)	62 sec.
VGG-A(2048, 128)	17 sec.	VGG-A(4096, 256)	58 sec.
VGG-E(4096, 128)	257 sec.	VGG-E(8192, 256)	1 hour
ResNet-18(4096, 256)	1 hour	ResNet-18(8192, 128)	2 hours
MobileNet-v1(2048, 128)	1.5 hour	MobileNet-v1(4096, 128)	4 hours

random searches. The extremely long search time prevents the exhaustive method traversing the whole search space for large-scale cases and it has to search only a number of random candidates.

5 CONCLUSIONS

In ReRAM-based CNN accelerators, crossbar allocation for layers, which affects both intra-layer and inter-layer parallelism, should be elaborated to maximum the performance and also to balance computation and communication. This problem is not comprehensively investigated in previous studies. The impact of communication (i.e., data access) on optimal crossbar allocation has never been studied. In this work, we build a mathematical framework to find near-optimal crossbar allocation schemes for ReRAM-based CNN accelerators. By modeling computation and communication behaviors in an abstract and unified way and solving the optimization problem through a dynamic programming based solver, our mathematical framework can obtain near-optimal crossbar allocation solutions quickly, without time-consuming exhaustive searches. We have demonstrated that intuitive heuristics trying to balance the weights or workloads for all layers are not optimal solutions. Instead, solutions obtained by our mathematical framework by comprehensively considering computation and communication are near-optimal and our method is much better than the crossbar allocation strategies of previous studies.

REFERENCES

[1] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-mei W. Hwu, John Paul Strachan, Kaushik Roy, and Dejan S. Milojicic. 2019. PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 715–731. https://doi.org/10.1145/3297858.3304049

[2] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. 269–284.

[3] Xiaoming Chen, Yinhe Han, and Yu Wang. 2020. Communication lower bound in convolution accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 529–541. https://doi.org/10.1109/HPCA47549.2020.00050

[4] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 609–622. https://doi.org/10.1109/MICRO.2014.58

[5] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 367–379. https://doi.org/10.1109/ISCA.2016.40

[6] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[7] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 27–39. https://doi.org/10.1109/ISCA.2016.13

[8] Teyuh Chou, Wei Tang, Jacob Botimer, and Zhengya Zhang. 2019. CASCADE: Connecting RRAMs to extend analog dataflow in an end-to-end in-memory processing paradigm. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 114–125.

[9] James Demmel and Grace Dinh. 2018. Communication-optimal convolutional neural nets. *arXiv* (2018). https://doi.org/10.48550/ARXIV.1802.06905

[10] Fabrice Devaux. 2019. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. 1–24. https://doi.org/10.1109/HOTCHIPS.2019.8875680

[11] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*. 580–587. https://doi.org/10.1109/CVPR.2014.81

[12] Peng Gu, Boxun Li, Tianqi Tang, Shimeng Yu, Yu Cao, Yu Wang, and Huazhong Yang. 2015. Technological exploration of RRAM crossbar array for matrix-vector multiplication. In *The 20th Asia and South Pacific Design Automation Conference*. 106–111. https://doi.org/10.1109/ASPDAC.2015.7058989

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[14] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[15] Shubham Jain and Anand Raghunathan. 2019. CxDNN: Hardware-software compensation methods for deep neural networks on resistive crossbar systems. *ACM Trans. Embed. Comput. Syst.* 18, 6 (Nov. 2019), 1–23. https://doi.org/10.1145/3362035

[16] Jin Hyun Kim, Shin-haeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, Bengseng Phuah, Jihyun Choi, Jinin So, YeonGon Cho, JoonHo Song, Jangseok Choi, Jeonghyeon Cho, Kyomin Sohn, Youngsoo Sohn, Kwangil Park, and Nam Sung Kim. 2021. Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for ML accelerators and beyond. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–26. https://doi.org/10.1109/HCS52781.2021.9567191

[17] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A fast and extensible DRAM simulator. *IEEE Comput. Archit. Lett.* 15, 1 (2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. 1097–1105.

[19] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sanghyuk Kwon, Je-Min Ryu, Jong-Pil Son, O. Seongil, Hak soo Yu, Hae-Sung Lee, Sooyoung Kim, Young-Cheol Cho, Jin Guk Kim, Jo-Bong Choi, Hyunsung Shin, Jin Hyun Kim, BengSeng Phuah, Hyoun Joo Kim, Myeongsoo Song, Ahn Choi, Daeho Kim, Sooyoung Kim, Eunhwan Kim, David Wang, Shin-Haeng Kang, Yuhwan Ro, Seungwoo Seo, Joonho Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. 2021. 25.4 A 20nm 6GB function-in-memory DRAM, based on HBM2 with a 1.2TFLOPS programmable computing unit using bank-level parallelism, for machine learning applications. *2021 IEEE International Solid-State Circuits Conference (ISSCC)* 64 (2021), 350–352.

[20] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O. Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware architecture and software stack for PIM based on commercial DRAM technology: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 43–56. https://doi.org/10.1109/ISCA52012.2021.00013

[21] Bing Li, Ying Wang, and Yiran Chen. 2020. HitM: High-throughput ReRAM-based PIM for multi-modal neural networks. In *2020 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 1–7.

[22] Weitao Li, Pengfei Xu, Yang Zhao, Haitong Li, Yuan Xie, and Yingyan Lin. 2020. Timely: Pushing data movements and interfaces in PIM accelerators towards local and in time domain. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 832–845. https://doi.org/10.1109/ISCA45697.2020.00073

[23] Bosheng Liu, Zhuoshen Jiang, Jigang Wu, Xiaoming Chen, Yinhe Han, and Peng Liu. 2021. F3D: Accelerating 3D convolutional neural networks in frequency space using ReRAM. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 571–576. https://doi.org/10.1109/DAC18074.2021.9586135

[24] Ximing Qiao, Xiong Cao, Huanrui Yang, Linghao Song, and Hai Li. 2018. AtomLayer: A universal ReRAM-based CNN accelerator with atomic layer computation. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC.2018.8465832

[25] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 14–26. https://doi.org/10.1109/ISCA.2016.12

[26] K. Simonyan and A. Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. *CoRR* abs/1409.1556 (2015).

[27] Gagandeep Singh, Mohammed Alser, Damla Senol Cali, Dionysios Diamantopoulos, Juan Gómez-Luna, Henk Corporaal, and Onur Mutlu. 2021. FPGA-based near-memory acceleration of modern data-intensive applications. *IEEE Micro* 41, 4 (2021), 39–48. https://doi.org/10.1109/MM.2021.3088396

[28] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. PipeLayer: A pipelined ReRAM-based accelerator for deep learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 541–552. https://doi.org/10.1109/HPCA.2017.55

[29] Tao Song, Xiaoming Chen, Xiaoyu Zhang, and Yinhe Han. 2021. BRAHMS: Beyond conventional RRAM-based neural network accelerators using hybrid analog memory system. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 1033–1038. https://doi.org/10.1109/DAC18074.2021.9586247

[30] Shibin Tang, Shouyi Yin, Shixuan Zheng, Peng Ouyang, Fengbin Tu, Leiyue Yao, JinZhou Wu, Wenming Cheng, Leibo Liu, and Shaojun Wei. 2017. AEPE: An area and power efficient RRAM crossbar-based accelerator for deep CNNs. In *2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 1–6. https://doi.org/10.1109/NVMSA.2017.8064475

[31] H.-S. Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T. Chen, and Ming-Jinn Tsai. 2012. Metal–Oxide RRAM. *Proc. IEEE* 100, 6 (2012), 1951–1970. https://doi.org/10.1109/JPROC.2012.2190369

[32] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 476–488. https://doi.org/10.1109/HPCA.2015.7056056

[33] Geng Yuan, Payman Behnam, Zhengang Li, Ali Shafiee, Sheng Lin, Xiaolong Ma, Hang Liu, Xuehai Qian, Mahdi Nazm Bojnordi, Yanzhi Wang, and Caiwen Ding. 2021. FORMS: Fine-grained polarized ReRAM-based in-situ computation for mixed-signal DNN accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 265–278. https://doi.org/10.1109/ISCA52012.2021.00029

[34] Qilin Zheng, Zongwei Wang, Zishun Feng, Bonan Yan, Yimao Cai, Ru Huang, Yiran Chen, Chia-Lin Yang, and Hai Helen Li. 2020. Lattice: An ADC/DAC-less ReRAM-based processing-in-memory architecture for accelerating deep convolution neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218590

[35] Zhenhua Zhu, Jilan Lin, Ming Cheng, Lixue Xia, Hanbo Sun, Xiaoming Chen, Yu Wang, and Huazhong Yang. 2018. Mixed size crossbar based RRAM CNN accelerator with overlapped mapping Method. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240825

[36] Zhenhua Zhu, Hanbo Sun, Kaizhong Qiu, Lixue Xia, Gokul Krishnan, Guohao Dai, Dimin Niu, Xiaoming Chen, Xiaobo Sharon Hu, Yu Cao, Yuan Xie, Yu Wang, and Huazhong Yang. 2020. MNSIM 2.0: A behavior-level modeling tool for memristor-based neuromorphic computing systems. In *GLSVLSI '20: Great Lakes Symposium on VLSI 2020, Virtual Event, China, September 7-9, 2020*, Tinoosh Mohsenin, Weisheng Zhao, Yiran Chen, and Onur Mutlu (Eds.). ACM, 83–88. https://doi.org/10.1145/3386263.3407647

Received 10 August 2022; revised 18 June 2023; accepted 24 October 2023