Efficient Mapping Space Exploration for a Reconfigurable Neural Accelerator

Zhongyuan Zhao*, Hyoukjun Kwon†, Sachit Kuhar‡, Weiguang Sheng*, Zhigang Mao*, and Tushar Krishna†
*Shanghai Jiao Tong University, †Georgia Institute of Technology, ‡Indian Institute of Technology Guwahati
zyzhao.sjtu@gmail.com, hyoukjun@gatech.edu, kuhar@iitg.ac.in
wgshenghit@sjtu.edu.cn, maozhigang@sjtu.edu.cn, tushar@ece.gatech.edu

***Abstract*—Deep learning accelerators have emerged to enable energy-efficient and high-throughput inference from edge devices such as self-driving cars and smartphones, to data centers for batch inference such as recommendation systems. However, the actual energy efficiency and throughput of a deep learning accelerator depends on the deep neural network (DNN) loop nest mapping on the processing element array of an accelerator. Moreover, the efficiency of a mapping dramatically changes by the target DNN layer dimensions and available hardware resources. Therefore, the optimal mapping search problem is a non-trivial high-dimensional optimization problem. Although several tools and frameworks exist for compiling to CPUs and GPUs, we lack similar tools for deep learning accelerators.**

To deal with the optimized mapping search problem in deep learning accelerators, we propose mRNA (mapper for reconfigurable neural accelerators), which automatically searches optimal mappings using heuristics based on domain knowledge about deep learning and an energy/runtime cost evaluation framework. mRNA targets MAERI, a recently proposed open-source deep learning accelerator that provides flexibility via reconfigurable interconnects, to run the unique mappings for each layer generated by mRNA. In realistic machine learning workloads from MLPerf, the optimal mappings identified by mRNA framework provides 15% to 26% lower runtime and 55% to 64% lower energy for convolutional layers and 24% to 67% lower runtime and maximum 67% lower energy for fully connected layers compared to simple reference mappings manually picked for each layer.

I. Introduction

Deep neural networks (DNNs) are now pervasive in both data centers [14, 24] and edge devices [1]. Modern DNNs have millions of parameters [35] and require billions of computations per layer. These extreme demands have led to an evolution of the target hardware that runs DNN inference from CPUs and GPUs to specialized hardware accelerators [3, 8, 11, 12]. These accelerators leverage data reuse opportunities to provide higher throughput / Watt than CPUs and GPUs.

Although DNN accelerators provide high energy-efficiency - a necessary property for mass deployment, a key challenge that hinders broad adoption is programmability. As is the case with CPUs and GPUs, programmability hides the complexity of the underlying hardware and allows software (applications/compilers/drivers) to automatically map computation and stage data movement within the accelerator. Moreover, in the case of DNN accelerators, this mapping directly affects the degree of data reuse, and data communication/staging patterns, and can have a huge impact on overall latency, throughput and energy. This is known as *dataflow*. and has received wide attention from the community [11, 26, 30, 33].

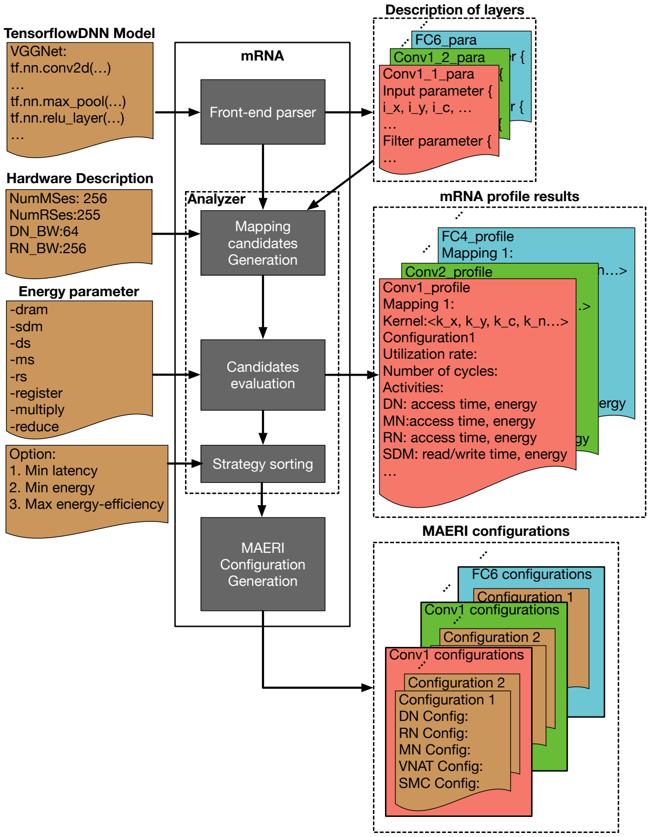


Figure 1: The mRNA tool flow

The programming model of many DNN accelerators today follows the same style as CPUs or GPUs - the computations to be mapped on DNN accelerators are defined in the form of coarse matrix-matrix or matrix-vector multiplications [14, 29]. The role of any mapper (or compiler) is to tile these matrices or vectors and map their computations spatially and/or temporally over the array. The dataflow implemented within the accelerator implicitly leads to a certain data communication/staging pattern from the scratchpad memories to the PEs, and between PEs for each mapping. Most DNN accelerators today are designed for one dataflow [8, 10–12, 21] which limits dataflow optimization opportunities

Authors Zhongyuan Zhao and Sachit Kuhar performed this research during their internships at the Georgia Institute of Technology.

based on (a) DNN layer dimensions or shapes and (b) available compute and memory within the accelerator.

There has been growing interest in designing accelerators that can support flexible dataflows [3, 30]. One such architecture, MAERI [3] supports arbitrary dataflows by leveraging light-weight, non-blocking, and reconfigurable tree-based interconnection network topologies within the accelerator. MAERI exposes fine-grained dataflow configurability to programmers via an abstraction known as *virtual neurons (VN)*, which is a temporary cluster of multipliers and adders that perform a multiply-accumulate operation to generate an output activation. MAERI can be configured to run any dataflow mapping via three features: (i) The multipliers have local scratchpad FIFOs. This allows data items to remain "stationary" for temporal reuse. (ii) The interconnects within MAERI support multicasts and local forwarding, enabling spatial reuse. (iii) The sizes of the VNs is completely configurable, and the substrate can also handle arbitrary sized VNs mapped simultaneously. This essentially allows support for arbitrary fine-grained tiling. MAERI can be configured layer-by-layer, or multiple times within a layer to handle folding/edge cases, or even cycle-by-cycle (subject to enough bandwidth on the control path [3]). The MAERI RTL is released as an open-source code-base [4] but there exists no methodology/tool to determine the right MAERI configuration for a target DNN layer and MAERI's microarchitectural parameters.

In this work, we propose mRNA (Mapper for Reconfigurable Neural Accelerator)[2] - a dataflow exploration and mapping engine that automatically searches through a suite of DNN mapping strategies for MAERI [3] and provides a set of energy- or throughput-optimal mappings. Figure 1 shows an overview. mRNA receives the neural network description, target hardware resources, and optimization goal (energy, runtime, etc.) as inputs and generates MAERI interconnection network configurations, which is equivalent to the machine code for the MAERI DNN accelerator, as outputs. At the heart of mRNA is a dataflow exploration engine that varies the mapping size of each DNN layer dimension and the order of nested DNN loops to search through a set of mapping candidates, compute the expected runtime, energy-efficiency, and compute unit utilization for each, and identify an optimal mapping strategy (dataflow) among the candidates.

The core contributions of this paper are as follows:

- We present an automatic mapper that searches and suggests a set of optimal DNN mappings over MAERI.
- A code generator that produces MAERI interconnection network configurations that specifies a target mapping.
- Case studies demonstrating the impact of different mappings on MAERI's performance and energy-efficiency with real DNN workloads from MLPerf.

mRNA can either be used as a stand-alone tool to explore optimal dataflows and mapping strategies for DNN kernels, or in conjunction with MAERI to map and run a DNN through RTL. mRNA has been open-sourced and is available for download[3].

The rest of the paper is organized as follows. Section II provides the necessary background on DNN computations and MAERI to understand this paper. Section III presents the mRNA mapping exploration engine. Section IV provides details on the actual framework. Section V demonstrates the impact of mapping strategies, and therefore the value of mRNA, across a suite of MLperf workloads and hardware configurations. Section VI discusses related work, and Section VII concludes.

II. Background

A. Computing patterns inside DNN models

Neurons, the fundamental unit of computation in DNNs, receives a certain number of input activations and generates one output activation. The number of input activations and weights depends on the neural network dimensions and algorithmic optimizations such as pruning. The computation inside a neuron consists of (1) Hadamard product that multiply each input activation and its corresponding weight value (element-wise multiplication), (2) reduction that sums up the element-wise product results (or, partial sums), and (3) activation of the reduced partial sums, which are non-linear functions such as sigmoid or ReLu that map a reduced partial sum (output of reduction) into a certain range (e.g., (-1, 1) for sigmoid).

A layer in DNN contains neurons with the same size whose size varies depending on the layer type and dimension. A neuron size of 3x3 is common in convolutional neural networks (CNN) [19, 35] based on neural net designer's choices, and 1x1 is common in LSTMs (Long short-term memories) [36] based on its definition. Each cell of an LSTM layer contains multiple neurons that compute input, forget, and output gate values, a state value, and the next hidden layer output.

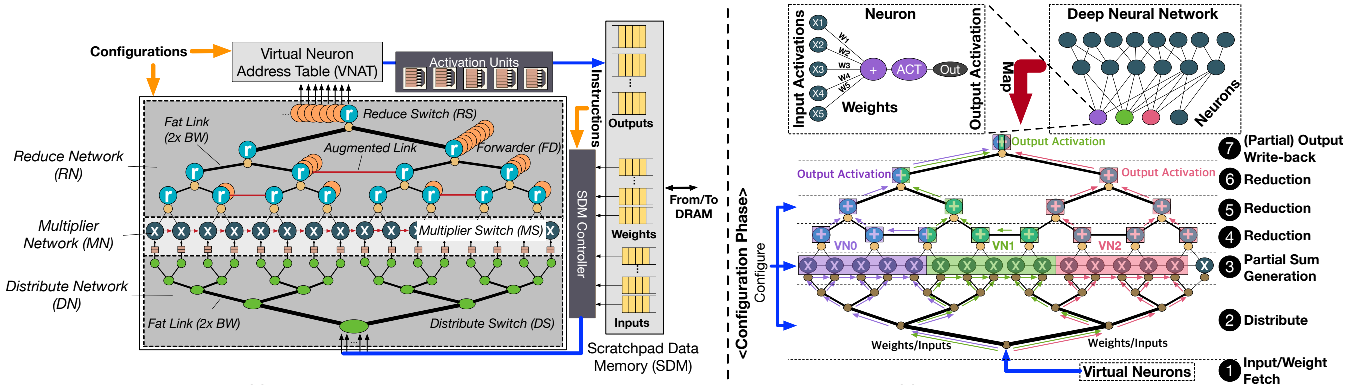
B. Communication patterns inside DNN accelerators

Communication patterns inside DNN accelerators can be classified into three categories [27]: distribution, local forwarding, and collection

Distribution. The computation of a neuron can be mapped over one or many processing elements (PEs) depending on the accelerator implementation. Neurons require weight and input activation tensor to generate an output. This leads to a phase of distribution of weights and input activations in the accelerator from a global scratchpad data memory (SDM) to local scratchpad memory in each PE, which is a one-to-many

[2] In biology, mRNA conveys genetic information from the DNA to the cells. Similarly, we envision our tool conveying mapping information from the DNN program to hardware processing elements.

[3] http://synergy.ece.gatech.edu/tools/maeri/mrna



(a) MAERI Microarchitecture

(b)Mapping DNNs over MAERI

Figure 2: MAERI [26] architecture and its two operation phases (configuration and operation) described with a mapping example.

communication. Depending on the mapping strategy, spatial data reuse (multicasting of a data) [26] is available during distribution, as long as the interconnection network between SDM and PEs provides multicasting capability.

Local forwarding. Based on the DNN layer type, some neurons can process partially overlapped input activations, which is known as sliding window behavior. To exploit this feature, PEs can forward data using neighbor-to-neighbor links rather than fetching data from the SDM that is further away on-chip, thereby reducing energy. This is known as spatial-temporal reuse [26] because data is reused in a different location at different time. This communication pattern is one of the core optimizations in all accelerators to increase energy efficiency.

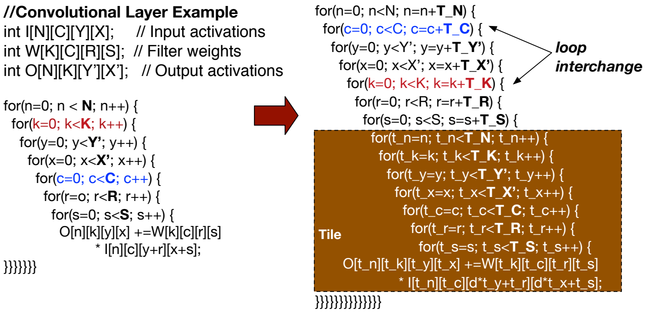
Collection. When each neuron produces a partial or full output activation depending on the mapping strategy, an accelerator needs to move the output activation to SDM. The communication pattern for moving output activations is collective from PEs to SDM (many-to-one). Because of the nature of many-to-one communication, the latency and throughput of collection completely depends on the effective bandwidth of the interconnection network from PEs to SDM.

C. *Target DNN architecture: MAERI*

MAERI [3] is an open-source reconfigurable DNN accelerator written in Bluespec System Verilog (BSV) that provides high compute unit utilization and performance with DNNs with both regular and irregular neuron sizes (sparse weight, cross-layer mapping, etc.). MAERI provides configurability via its three reconfigurable interconnection networks - distribution network (DN), multiplier network (MN), and reduce network (RN).

1) Microarchitecture: As shown in Figure 2, MAERI consists of the three networks (DN, MN, RN), a virtual neuron address table (VNAT), activation units, global scratchpad data memory (SDM), and SDM controller. The DN nodes are simple 1:2 switches, the MN nodes are multipliers with tiny 2:2 switches (multiplier switches - MS), and the RN nodes are adders with tiny 2:3 switches (reduction switches: RS). The topology of the DN, MN and RN is a fat-binary tree, linear, and augmented-reduction tree (ART [3]), respectively. The ART is a fat-binary adder with additional forwarding links between two adjacent nodes in the same level that do not share a common parent node, marked in red in Figure 2 (a). ART provides non-blocking reduction for any size of virtual neurons mapped on the MN, enabling high compute unit utilization for neurons with irregular sizes. The bandwidth of both the DN and RN is design-time configurable. The bandwidth at the root of the DN (i.e., number of unique inputs/weights that can be read) is set to match the output bandwidth of the SRAMs, and it tapers down by a factor of two at every level till it becomes one. Similarly for the ART.

2) Operations Phases: MAERI comprises of three operation phases - mapping strategy exploration (MSE), configuration, and execution. During the *MSE phase* - which is the focus of this work - a compiler (or user) determines the best mapping strategy for given DNN layer dimensions (number of filters, filter height/width, number of channels, and so on) and hardware resources (number of MSes, DN/RN bandwidth). We discuss the MSE phase in detail in Section III. During the *configuration phase*, a controller inside MAERI sends configuration signals to the switches in DN, MN and RN to implement the mapping strategy determined in MSE phase. The configuration determines the size of each virtual neuron (VN) over MAERI, as Figure 2 (b) shows. The VN is the key computation primitive in MAERI. The sizes of each VNs can be layer-specific (e.g., dense convolutions) or neuron specific (e.g., sparse convolutions). In this work, we assume dense DNNs, where all VNs that are currently mapped have the same size, as Figure 2(b) shows. MAERI can also handle VNs of different sizes (for sparsity) but that is beyond the scope of our tool at the moment. The VN configurations need to be updated every time the MSE provides a new configuration, which depends on the DNN layer type and dimensions. Correspondingly, the control signals for configurations can be sent via a low-bandwidth scan chain or a high-bandwidth control plane, depending

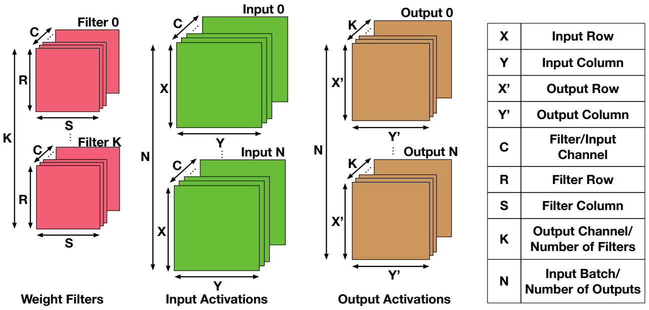


```
//Convolutional Layer Example
int I[N][C][Y][X];     // Input activations
int W[K][C][R][S];  // Filter weights
int O[N][K][Y'][X'];   // Output activations

for(n=0; n < N; n++) {
 for(k=0; k<K; k++) {
  for(y=0; y<Y'; y++) {
   for(x=0; x<X'; x++) {
    for(c=0; c<C; c++) {
     for(r=o; r<R; r++) {
      for(s=0; s<S; s++) {
       O[n][k][y][x] +=W[k][c][r][s]
                * I[n][c][y+r][x+s];
}}}}}}}
```

```
for(n=0; n<N; n=n+T_N) {
 for(c=0; c<C; c=c+T_C) {
  for(y=0; y<Y'; y=y+T_Y') {
   for(x=0; x<X'; x=x+T_X') {
    for(k=0; k<K; k=k+T_K) {
     for(r=0; r<R; r=r+T_R) {
      for(s=0; s<S; s=s+T_S) {
       for(t_n=n; t_n<T_N; t_n++) {
        for(t_k=k; t_k<T_K; t_k++) {
         for(t_y=y; t_y<T_Y'; t_y++) {
          for(t_x=x; t_x<T_X'; t_x++) {
           for(t_c=c; t_c<T_C; t_c++) {
            for(t_r=r; t_r<T_R; t_r++) {
             for(t_s=s; t_s<T_S; t_s++) {
     O[t_n][t_k][t_y][t_x] +=W[t_k][t_c][t_r][t_s]
               * I[t_n][t_c][d*t_y+t_r][d*t_x+t_s];
}}}}}}}}}}}}}}
```

Figure 3: Loop transformations (tiling and loop interchange) of a convolution loop nest. ”d” stands for stride.



X	Input Row
Y	Input Column
X'	Output Row
Y'	Output Column
C	Filter/Input Channel
R	Filter Row
S	Filter Column
K	Output Channel/ Number of Filters
N	Input Batch/ Number of Outputs

Figure 4: Loop variable convention for a convolutional layer.

on the implementation. Finally, during the *execution phase*, weight and input activation tensors are first distributed to the MN via DN, the MN computes partial sums, and the RN spatially reduces these to partial or full outputs. Each VN generates a partial/full outputs that are sent via activation units to the output buffers in the SDM. More details on the configuration and execution phases are provided in the MAERI paper [3] and are not the focus of this work.

III. Mapping Space Search

Almost all the layers inside DNN models can be represented as a multi-level nested loops, and those loops rarely carry dependencies other than reduction dependency, which implies abundant parallelism to exploit. Moreover, standard loop transformation methods such as loop interchange and tiling can be applied to the loop, as shown in Figure 3. The effect of such loop transformations are significant in deep learning accelerators, which dramatically changes the throughput, latency, and energy efficiency, because they influence the data communication pattern among SDM, DN, MN, RN, and DRAM [26]. We term each transformed loop nest as a *mapping* and discuss the impact of mappings in the following sections.

A. Mapping Taxonomy

Instead of having full version of loops in each layer, we abstract the loops into multiply and accumulate (MAC) instances over multi-dimensional weights and inputs. For example, convolutional layers in CNN are MAC operations with four-dimensional weights, inputs, and outputs: W(R, S, C, K), I(X, Y, C, N) and O(X’, Y’, K, N), the meaning of each dimension is shown in Figure 4. We use the convention of W/I/O followed by four dimension parameters in braces to specify the dimension of a convolutional layer. Because some of the indices overlap in convolutions (e.g., channel of input and weights), total number of parameters in convolution is seven. Based on the seven convolutional parameters, we define seven parameters to specify a specific mapping tile mapping on MAERI’s hardware resources. These are presented in Table I. From MAERI’s perspective, the mapped tile parameters specify the VN size (i.e., number of MAC operations in each VN) and number of VNs, as Table I shows.

Table I: mRNA Seven Mapping Parameters

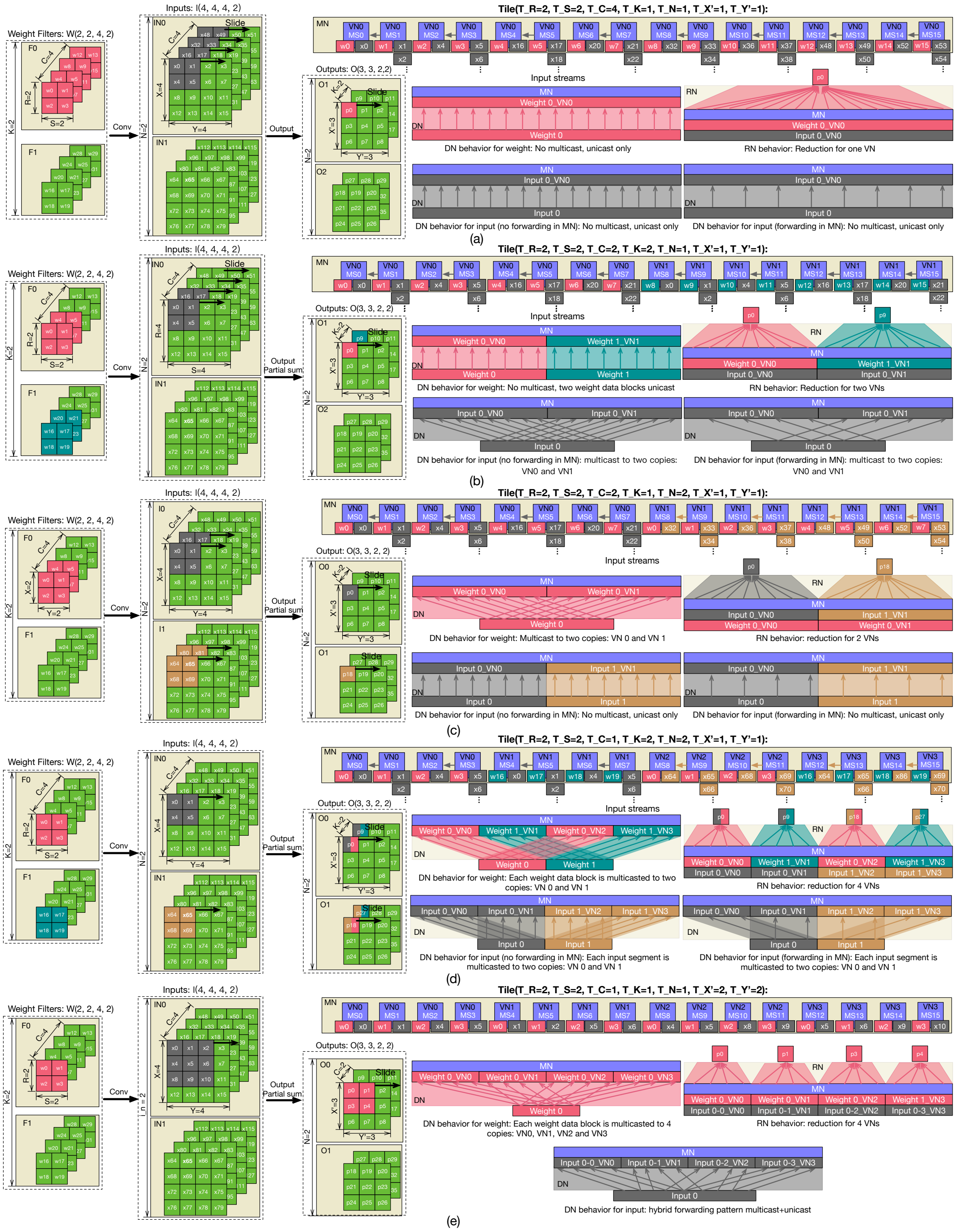
Symbol	Description
T_R	The number of mapped rows of inputs and weights in a tile
T_S	The number of mapped columns of inputs and weights in a tile
T_C	The number of mapped input and weight channels in a tile
T_K	The number of mapped filters in a tile
T_N	The number of mapped input batches in a tile
T_X’	The number of mapped rows of outputs in a tile
T_Y’	The number of mapped columns of outputs in a tile
VN Size	$T\_R \times T\_S \times T\_C$.
Num VNs	$T\_K \times T\_N \times T\_X' \times T\_Y'$

In the rest of the the paper, we use the convention of “Tile” followed by the seven mapping parameters in a parenthesis to specify a tile or the size of mapped volume for each dataclass: Tile(T_R, T_S, T_C, T_K, T_N, T_X’, T_Y’).

B. The Impact of Mapping

Figure 5 shows five mappings of a custom convolutional layer on 16 MSes of MAERI. In this layer, the parameters of the filters, inputs, and outputs are W(2, 2, 4, 2), I(4, 4, 4, 2) and O(3, 3, 2, 2). We discuss each mapping, and its potential impact on reuse and communication bandwidth (which in turns affects performance and energy), next.

The mapping in Figure 5 (a) computes over the whole F0 filter (2x2x4) and input IN0. In each control step (i.e., iteration) during the tile execution, MAERI performs MAC operation over 16 weight and input elements and generates a partial output (p0). In the DN, 16 weights (weight 0) and input elements are uni-casted to the MN. In the MN, some input elements can be reused by forwarding the value to the adjacent MSes as the filter window slides along the row of the input. Therefore, if the MN enables forwarding data between adjacent MSes, only a subset of input elements need to be sent through the DN in the subsequent control steps; in this example, only eight input elements are unicast through the DN. In the RN, since one VN produces a full output, one mapping configuration is sufficient to generate the full output. However, this mapping does not exploit spatial reuse



Weight Filters: W(2, 2, 4, 2)
Inputs: I(4, 4, 4, 2)
Outputs: O(3, 3, 2, 2)
Conv
Output
Output Partial sum
Slide
Tile(T_R=2, T_S=2, T_C=4, T_K=1, T_N=1, T_X'=1, T_Y'=1):
Input streams
DN behavior for weight: No multicast, unicast only
RN behavior: Reduction for one VN
DN behavior for input (no forwarding in MN): No multicast, unicast only
DN behavior for input (forwarding in MN): No multicast, unicast only
(a)
Tile(T_R=2, T_S=2, T_C=2, T_K=2, T_N=1, T_X'=1, T_Y'=1):
DN behavior for weight: No multicast, two weight data blocks unicast
RN behavior: Reduction for two VNs
DN behavior for input (no forwarding in MN): multicast to two copies: VN0 and VN1
DN behavior for input (forwarding in MN): multicast to two copies: VN0 and VN1
(b)
Tile(T_R=2, T_S=2, T_C=2, T_K=1, T_N=2, T_X'=1, T_Y'=1):
DN behavior for weight: Multicast to two copies: VN 0 and VN 1
RN behavior: reduction for 2 VNs
DN behavior for input (no forwarding in MN): No multicast, unicast only
DN behavior for input (forwarding in MN): No multicast, unicast only
(c)
Tile(T_R=2, T_S=2, T_C=1, T_K=2, T_N=2, T_X'=1, T_Y'=1):
DN behavior for weight: Each weight data block is multicasted to two copies: VN 0 and VN 1
RN behavior: reduction for 4 VNs
DN behavior for input (no forwarding in MN): Each input segment is multicasted to two copies: VN 0 and VN 1
DN behavior for input (forwarding in MN): Each input segment is multicasted to two copies: VN 0 and VN 1
(d)
Tile(T_R=2, T_S=2, T_C=1, T_K=1, T_N=1, T_X'=2, T_Y'=2):
DN behavior for weight: Each weight data block is multicasted to 4 copies: VN0, VN1, VN2 and VN3
RN behavior: reduction for 4 VNs
DN behavior for input: hybrid forwarding pattern multicast+unicast
(e)

DN_BW input req	16, 8
DN_BW weight req	16
RN_BW req	1
Control steps	36
DN input communication	64, 32
DN weight communication	64
MN communication (reg+forwarding)	32
RN communication	30

DN_BW input req	8, 4
DN_BW weight req	16
RN_BW req	2
Control steps	41
DN input communication	64, 32
DN weight communication	64
MN communication (reg+forwarding)	32
RN communication	28

DN_BW input req	16, 8
DN_BW weight req	8
RN_BW req	2
Control steps	41
DN input communication	64, 32
DN weight communication	64
MN communication (reg+forwarding)	32
RN communication	28

DN_BW input req	8, 4
DN_BW weight req	8
RN_BW req	4
Control steps	45
DN input communication	56, 28
DN weight communication	64
MN communication (reg+forwarding)	32
RN communication	24

DN_BW input req	9
DN_BW weight req	4
RN_BW req	4
Control steps	73
DN input communication	60
DN weight communication	56
MN communication (reg+forwarding)	32
RN communication	24

Figure 5: Examples of mapping strategy 1 (a), 2 (b), 3 (c) and 4 (d), 5(e). The four dimensional parameters in Weight Filter: W()/Input: I()/Output: O() represents the height, width, channel number and the number of filter/input/output. The seven dimensional parameters witin Tile() is row, column, channel, filter number and input number, output_row and output_Column The arrows inside DN and RN represents the forwarding path of each element. In table on the right most, req means requirement and the communication is the number within single control step (i.e, iteration), and the unit of bandwidth is the number of deliverable data points per cycle.

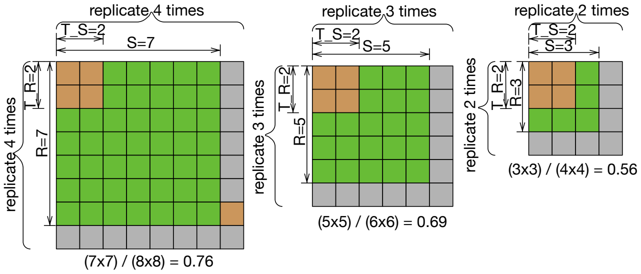


Figure 6: The overall utilization rate of selecting $T\_R = T\_Y = 2$ when R and S are both equal to 7, 5, 3 respectively. Green area shows the filter weights and the brown area in the upper-left shows a tile. The gray area around the green area shows invalid coverage of tiles, which is out-of-bound that causes significant underutilization as presented at the bottom of each example.

via multicasting in DN, thus requiring high bandwidth and dense data communication in the DN.

The mapping in Figure 5 (b) performs MAC over two 2x2x2 weights and one 2x2x2 input elements. In the DN, correspondingly, eight weights in filter 0 (weight 0) and filter 1 (weight 1) are unicasted with tag VN_0 and VN_1, which indicates their corresponding VN ID. For the first control step, eight unique input elements in input 0 are multicasted through DN to both VN0 and VN1; in the subsequent iterations, four of the inputs are forwarded locally (spatio-temporal reuse) and four new input elements are multicasted through the DN. The forwarding opportunities in the MN exist since the mapping exploits sliding window over the input. Because this mapping multicasts inputs in DN, DN requires half bandwidth compared to Figure 5 (a) (16 to 8 when forwarding in MN is inactive and 8 to 4 when forwarding in MN is active). In the RN, the two VNs generate two partial outputs (p0 and p9), not full outputs, which requires additional configuration phases to accumulate partial outputs to generate full outputs.

The mapping in Figure 5 (c) performs MAC over a 2x2x2 weight and two 2x2x2 input tiles; here weight elements are multicasted but input elements are not. In the DN, the bandwidth requirement for weight is larger than that of Figure 5 (b) because this mapping does not utilize multicast in DN for inputs. In the MN, input element forwarding is available for every other MS because of input sliding window. In the RN, two VNs generate two partial outputs (p0 and p19), which also requires additional configuration phases.

The mapping in Figure 5 (d) performs MAC over two 2x2x1 weight and two 2x2x1 input tiles, which generates four partial outputs in RN. In the DN, the bandwidth requirement is reduced compared to Figure 5 (c) for both the weight and input because both weight and input elements are multicasted. In the MN, forwarding is available for every other MS. In the RN, four VNs generate partial outputs that covers less number of partial sums compared to previous examples, which requires more number of control steps.

The mapping in Figure 5 (e) performs MAC over one 2x2x1 weight and four 2x2x1 input tiles: (x0, x1, x4, x5), (x1, x2, x5, x6), (x4, x5, x8, x9) and (x5, x6, x9, x10). In the DN, although weight elements are fully multicasted to all the VNs, input elements are partially multicasted, which introduces irregular multicasts and unicasts. In the MN, because adjacent multipliers require different input elements, forwarding is unavailable. In the RN, four VNs generate four partial outputs, which require additional control steps like Figure 5 (d) to generate full outputs.

As we observed in the examples, different mappings lead to different computation and communication patterns in hardware, which results in different throughput and energy efficiency. Therefore, exploring mapping options to identify the most throughput-optimized and energy-efficient one is critical to maximize the benefits of a flexible deep learning accelerator like MAERI. To identify such mappings, mRNA's mapping tool searches through all the potential mapping candidates and identify best mappings based on the user options (such as optimization goal). We discuss the mapping tool in the following subsection.

C. Mapping Space Search

The mapping algorithm includes three main phases: (1) choose the possible mapping candidates, (2) make a comprehensive evaluation on each candidate including a cycle-level estimate of performance, normalized energy, and energy-efficiency, (3) generate the configuration for MAERI according to the specified mapping strategy.

1) Candidates search: In the candidates search phase, the mapper searches mappings that utilizes available compute units as much as possible while satisfying the inequality in Equation 1:

$$\begin{aligned} T\_R \times T\_S \times T\_C \times T\_K \times T\_N \times T\_X' \times T\_Y' \leq N_{ms} \\ \text{where } T\_R \leq R,\ T\_S \leq S,\ T\_C \leq C,\ T\_K \leq K, \\ T\_N \leq N,\ T\_X' \leq X',\ T\_Y' \leq Y' \end{aligned} \tag{1}$$

N_{ms} represents the available computational resources, which is the number of MSes in MN. Because of the high dimensionality (7D), the mapping space based on the permutations of the loop and the mapping parameters of each loop is huge. For example, the second CONV layer of VGG16 [35] with batch size 32 has layer dimensions of I(224, 224, 64, 32), W(3, 3, 64, 64), O(224, 224, 64, 32). If we map this layer onto MAERI with 256 MSes, the number of all the possible mapping parameters that satisfy inequality Equation 1 is 71107. If we scale up the computational resources (number of MSes) to 1024, the search space will increase to 531517. It is impractical to evaluate the efficiency of all these mapping strategies in detail due to the large time complexity. We demonstrate that it is possible to prune down the search space by identifying mappings that are guaranteed to be inefficient compared to others.

To reduce the search space, we develop heuristics from the following insights.

MS utilization and edge conditions in mapping. Because the number of utilized MSes is determined based on the size of mapped VNs (e.g., if we map VNs with size 3 over a MS array with 16 MSes, only 15 MSes can be utilized), the mapping parameters that determine the size of VNs (T_R, T_S, and T_C) determine the peak utilization of MSes. Thus VN sizes that are not divisible by the number of MSes can be pruned away without a full evaluation. Similarly, when choosing appropriate tile sizes, the dimension of the filters (R, S, and C) needs to be considered to account for the underutilization when a tile encounters out-of-bounds at edges, as presented in Figure 6. To prevent such underutilization, we select mapping parameters, T_R, T_S, and T_C, that evenly divide each dimension, R, S, and C, respectively. We also apply the same constraints to input mapping parameters. By applying these edge conditions, we can significantly reduce the search space.

Batch size in inference.. In most inference scenarios (forward pass), the batch size (N) is one. Therefore, we can assume that the corresponding mapping parameter, T_N, is also one, which removes one dimension of the 7D mapping search space.

Mapping fully-connected and LSTM.. Fully-connected (FC) and LSTM layers require matrix multiplication, which can be viewed as convolution with reduced dimensions ($C = K = 1$, $Y = C = 1$, $X' = K = 1$, $X = S$, $R = Y'$). Therefore, the number of mapping parameters to determine decreases to two (T_R and T_S), and this significantly reduces the mapping search space.

Partial and full outputs. The number of MACs for each neuron and virtual neuron is $R \times S \times C$ and $T\_R \times T\_S \times T\_C$, respectively. If $T\_R < R$, $T\_S < S$ or $T\_C < C$, each VN generates a partial output. The number of partial outputs per a full output ($N_{partial}$) is as follows:

$$N_{partial} = \left\lceil \frac{R}{T\_R} \right\rceil \times \left\lceil \frac{S}{T\_S} \right\rceil \times \left\lceil \frac{C}{T\_C} \right\rceil \tag{2}$$

That is, mRNA needs to generate another configuration to accumulate partial outputs to produce a full output, which results in the reconfiguration of the MAERI components (DN, MN, RN, and VNAT), which is expected to add latency and synchronization overheads. Furthermore, when the number of MSes is insufficient to cover $N_{partial}$, mRNA needs to generate an additional configuration for spatial folding. To prevent such additional reconfiguration overheads, the mapping candidates are chosen to try and compute full outputs within one control step, or satisfy $N_{partial} \leq N_{ms}$.

2) Evaluation on mapping candidates: To determine the best mapping, mRNA evaluates each mapping candidate using the following metrics: the average MS utilization, runtime, and energy consumption.

Average MS utilization. To evaluate the average MS utilization, we compute the total number of utilized MSes in each control step (each pipeline stage in MAERI presented in Figure 5 (b)) and divide it by the total number of MSes multiplied by the number of compute steps, as presented in Equation 3.

$$\frac{\sum_{cs=1}^{CS_{total}} U_{cs}}{CS_{total} \times N_{ms}} \tag{3}$$

where U_{cs} is the number of utilized MSes at control step cs.

Runtime. Although MAERI architecture allows non-blocking computation, serialization delay can be encountered when the bandwidth of DN and RN is not sufficient to support the DN and RN traffic for a mapping. mRNA identifies the bandwidth requirement of a mapping and uses it to compute possible serialization delay that adds up the total runtime.

Energy consumption. To evaluate energy consumption of the mapping candidates, we performed a layout of MAERI's RTL [4] using TSMC 28nm technology and extracted the relative energy consumed by multiplication, addition, buffer reads, and the various communication networks. We apply the extracted energy parameters to the activity counts mRNA generates, and compute the relative energy consumption of each mapping candidate.

IV. mRNA Framework

Figure 1 shows an overview of the mRNA toolflow. mRNA receives a DNN model written in Tensorflow [6], hardware resource description (number of MSes, the bandwidth of DN and RN, and so on), energy parameters (mult/add op energy, DN/RN traversal energy, local buffer/SDM access energy, etc.), and optimization options such as optimization goal (latency, throughput, or energy) as input. The mRNA framework analyzes the received inputs and generates optimized MAERI mappings for the optimization goal from the user and reports corresponding costs (relative energy consumption, the number of computations, communication, and buffer activities, and so on) as output. Figure 1 shows a snapshot of the outputs. The mRNA framework consists of a front-end parser, analyzer, and MAERI component configuration generator, which we discuss in the following subsections.

A. Front-end parser

The front-end parser receives the DNN model in Tensorflow format as input and extracts the DNN dimensions of each layer, which are dimension parameters discussed in Section III-B. The parser can also analyze FC layer and LSTM models; for LSTM, it extracts the dimension parameters for all the hidden units. The parser generates an intermediate dimension parameter file in a format described in Figure 1. The format has four separate sections (input, filter, output, and hidden paramter sections) that describes dimension parameters in each data class. This file can also be provided directly as an input by the user.

Table II: Dimension Parameters of Layers

Layer	Inputs I	Weight Filters W	Outputs O
CB3a_2	I(28,28,128,1)	W(3,3,128,128)	O(28,28,128,1)
CB5_2	I(7,7,512,1)	W(3,3,512,512)	O(7,7,512,1)
1x1red	I(7,7,832,1)	W(1,1,832,32)	O(7,7,32,1)
FC1	I(4096,1,1,1)	W(4096,4096,1,1)	O(4096,1,1)
FC2	I(2048,1,1,1)	W(2,2048,1,1)	O(2,1,1,1)
embed	I(1,1,138000,1)	W(1,1,138000,32)	O(1,1,32,1)

B. Analyzer

The analyzer receives the intermediate dimension parameter file from the front-end parser, hardware resource description, energy parameters, and optimization option as input, as described in Figure 1. Using the inputs, the analyzer generates mapping candidates and identifies optimized mappings using the mapping evaluation methods discussed earlier in Section III-C. The analyzer generates a set of optimized mappings and directly passes them to the MAERI configuration generator. For each mapping for each layer, it also generates a report file with the following information:

- **MS utilization:** The peak utilization, the average utilization for each configuration, and the average utilization across all the configurations.
- **Latency:** The number of cycles to run MAERI for all configurations.
- **DN activities:** The number of DN traversals, the energy consumption for each configuration, and the energy consumption for all the configurations.
- **MN activities:** The number of MN forwarding link traversals, the number of multiplications, the energy consumption for each configuration, and the energy consumption for all the configurations.
- **RN activities:** The number of RN traversals, the number of additions, the energy consumption for each configuration, and the energy consumption for all the configurations.
- **SDM activities:** The number of SDM accesses, the energy consumption for each configuration, and the energy consumption for all the configurations.
- **DRAM activities:** The number of DRAM accesses, the energy consumption for entire computing phase of a layer.

C. MAERI configuration generator

The MAERI configuration generator receives the optimized mappings from the analyzer as input and generates the configurations for each component (DN, MN, RN, SDM, VNAT, etc.) as output. The generated outputs are essentially the multiplexer select lines for all MAERI switches, and can directly loaded into MAERI to begin the configuration phase to setup the VNs (Section II-C2).

V. Evaluation

A. Methodology

We evaluate mappings generated by mRNA over a mix of DNN layers as shown in Table II. We select three convolutional layers from MLPerf [5]: CB3a_2 and CB5_2 from ResNet [19] and 1x1red from GoogLeNet [34]; we select two fully-connected layers: FC1 from Alexnet [25] and FC2 from Seq-CNN [23]; and we select one embedding layer from a MLP for neural collaborative filtering [20]. We evaluate the runtime, MS utilization rate, the interconnection network (DN, MN, and RN) activities, and energy consumption of mRNA mappings for the six selected layers. For convolutional layers, we only present results from the worst and best mappings because CNN has many possible mappings (e.g., mRNA generated 71107 mappings for VGG16-conv2). In the interest of space, we assume inference in all our experiments, which means that the T_N parameter (i.e., batch size) in the mappings remains 1.

B. Runtime

Full Bandwidth. Figure 7 present the runtime, MS utilization, energy consumption breakdown, and the number of the DN, MN and RN activities for each of our evaluation layers. We normalize all results to a *baseline* mapping strategy, which we define as Tile(T_R, T_S, 1, T_K, 1, 1, 1). In other words, only one channel of each filter is mapped over the MSes. For hardware configuration, we use 512 MSes and DN/RN with bandwidth of 512, which means 512 new data items can be delivered and received at the same time in DN and RN. This is an extreme data point, but demonstrates the impact of the mapping given no constraint on bandwidth. We will explore the effect of bandwidth later in the evaluations. For ResNet-CB3a_2 and ResNet-CB5a_2, the baseline mapping requires the longest runtime because low utilization at K edges. The utilization at K edges in the baseline mapping is 2x2x(128 mod 56)/512 = 64/512 = 12.5% as T_K and K are 56 and 128, respectively. Such low utilization significantly degrades the average utilization, which introduced inefficiency in the baseline. For GoogleNet-1x1red layer, the worst mapping is not the base line but a mapping that uses $T\_C = 512$, as presented Figure 7 (c). The average utilization of the worst mapping ($T\_C = 512$ and other mapping parameters are all 1) is only 81% because of the edge condition on C, like ResNet cases. For FC and embedding layers, all the mappings maintained high utilization but required dramatically different runtimes, as Figure 7 (d), (e), and (f) shows. This is because some mappings generate partial outputs like the example in Figure 5 (d), which requires additional control steps to generate full outputs.

Limited Distribution Bandwidth. Figure 8 shows that the effect of DN bandwidth over the runtime of mappings of (a) ResNet-CB3a_2 and (b) GoogleNet-1x1red layers, normalized to the runtime of the left-most mapping. For ResNet-CB3a_2 with the DN bandwidth of 16, the best mapping for runtime is Tile(3,3,4,14,1,1,1), which is the mapping in the middle. However, when the DN bandwidth increases to 128, the best mapping for runtime is Tile(3,3,8,7,1,1,1), the right-most mapping in the plot. This is because the

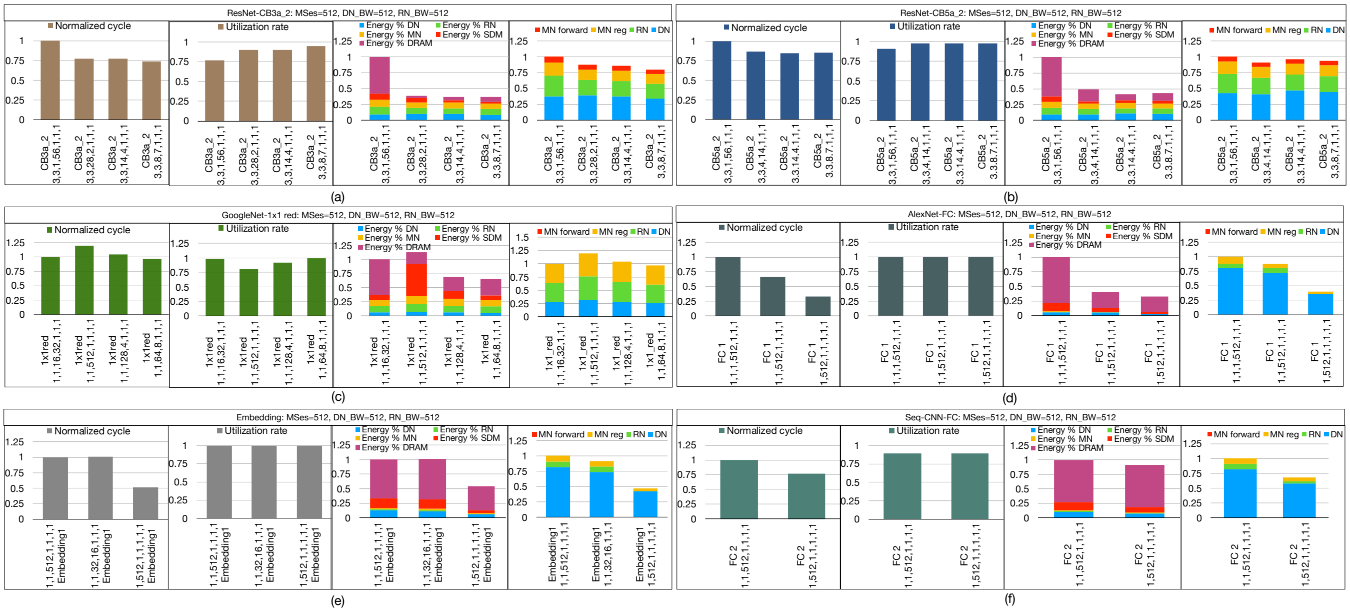


Figure 7: Mapping-Space Exploration with mRNA. We plot the runtime, utilization, energy consumption, and interconnect activity across the DNN layers listed in Table II for the top 2-4 mapping strategies generated by mRNA.

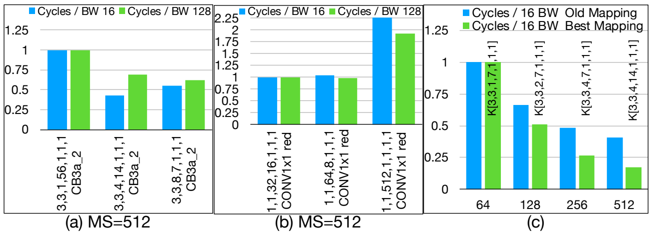


Figure 8: Impact of fixed distribution network bandwidth on runtime for (a) ResNet CB3a_2 with MS=512, (b) Google LeNet 1x1 with MS=512, and (c) ResNet CB3a_2 with increasing number of MSes.

mapping Tile(3,3,4,14,1,1,1) requires large DN bandwidth so DN with bandwidth 16 becomes the bottleneck of the entire pipeline. Mappings with multicasting of inputs (i.e., T_K is large) reduces bandwidth requirement and mitigates such bottlenecks from the DN. For GoogleNet-1x1red, we can observe the similar trend but the effect is not as significant as ResNet-CB3a_2 because the multicasting factor is larger in GoogleNet-1x1red based on 1x1 convolution window.

In Figure 8(c) we show how a fixed bandwidth (of 16) and mapping influences scalability. All blue bars use the baseline mapping, while the green bars plot the best mapping strategy. We can see that doubling MSes reduces runtime for the blue bars due to more number of MSes but by less than a factor of 2x due to serialization. The best mapping strategy is different everytime as it is the one that requires least DN bandwidth.

C. Compute unit (MS) utilization

The utilization of computational resources (MS in MAERI) is one of the key aspects that determine the efficiency and throughput of a deep learning accelerator. In MAERI, utilization depends on the number of VNs we can simultaneously run over the MSes, which in turn depends on VN size and number of MSes, as Section III-A discussed[4]. When the VN_size is not divisible by the number of MSes, there is underutilization. The utilization naturally directly affects runtime. For example, the lowest MS utilization in Figure 7 (a) is 76%, and its corresponding mapping requires the longest runtime. As we discussed in the previous subsection, underutilization at edges degrades the utilization of Figure 7(a). The impact of low utilization at edges to the average utilization differs by the dimension and mapping factor. For example, the lowest utilization of Figure 7 (b) is $512/(56 \times 10) = 0.91$, whose gap is much smaller than that of Figure 7(a). Also, the lowest utilization of Figure 7(c) is C $(832/(512 \times 2) = 0.81$, which is lower than that of Figure 7(b) because of a large T_C (512).

D. MAERI Interconnection Network Activities

The right-most plots in Figure 7 (a-f) show the number of traversals across the DN, MN and RN. We find the number of activities in DN, RN, and MN almost identical in Resnet convolutional layers. For Googlenet_1x1red layer (which has a convolutional window size of 1x1), and all non-convolutional layers, the number of MN forwards is zero as there is no sliding window behavior to exploit spatio-temporal reuse.Unlike convolutional layers, FC layers require heavy DN activities compared to those of RN and MN. This is due to the small multicasting factors (mostly 1) in DN on FC layers, which leads to more number of unicast in DN.

[4] We currently do not support mapping of partial VNs except for the case when the VN size is greater than the number of MSes.

E. Energy

The DRAM access energy dominates in total energy consumption, as we can observe in the third plot of each layer in Figure 7. One of sources of exhaustive DRAM accesses is partial outputs generated from virtual neurons. For example, for layers with large number of partial sums to accumulate for a full output ($T\_R \times T\_S \times T\_C$) and mappings that generate partial outputs, not full outputs, may store partial outputs in DRAM and accumulate them after MAERI computes all the partial outputs. The number of extra DRAM accesses in this cases depends on the on-chip storage size. Therefore, mappings whose VNs generate full outputs or partial outputs with more number of partial sums accumulated can significantly reduce DRAM energy.

F. Discussion

Based on the above analysis, we observe that the best mappings that mRNA finds have the following features:

(1) High computational resource (MS) utilization: High MS utilization leads to small runtime and energy.

(2) High multicast factors in DN: Multicasts (or, spatial data reuse) in DN reduces the number of DN activities by merging unicasts into one multicast, which reduces energy as well as runtime.

(3) Small number of partial outputs for each full output: Small number of partial outputs for a full output decreases DRAM accesses and requires smaller extra control steps to accumulate partial outputs to full outputs.

VI. RelatedWorks

Mappers for DNN Accelerators. XLA [2] is the Tensorflow [7] back-end compiler that generates configurations for Google TPU (Tensor Processing Unit) [24]. FP-DNN [16] and DNN Weaver [33] include mappers that respectively takes DNNs written in TensorFlow and Caffe [22] as input and generates DNN hardware RTLs targeting FPGAs that run with mappings the mapper generated as output. MAESTRO [26] is recent DNN dataflow design-space exploration analytical model for an abstract spatial accelerator. In contrast, mRNA is a mapper performing automatic search for optimal dataflows per layer given a specific hardware configuration of MAERI.

Mappers for CGRAs. REGIMap [17], RAMP [13], EPIMap [18] and Resource-saving [37] are mappers for Coarse-Grained Reconfigurable Architectures (CGRA). They use the modulo scheduling based algorithm to perform software pipelining of the loop body and try to minimize the initiation interval (II) between iterations. The mapper proposed in Nowatzki et al. [32] maps applications onto a dynamic CGRA [15] based on a stream-dataflow accelerator [31].

Mappers for GPUs and others. Legion [9] is a programming model and a runtime system that supports parallel architectures including GPUs. It allows users to specify mappings of their application and also provides a default mapper based-on greedy algorithm on memory size and bandwidth. A GPU architecture-aware automatic mapper was proposed in Lee et al. [28]. This GPU mapper analyzes parallel program patterns in the target program and construct a mapping space considering GPU architecture. The mapper searches constructed mapping space based on the constraints from architecture and degree of parallelism scores, which maximizes throughput.

VII. Conclusion

In this paper, we presented a mapping framework, mRNA, for automatically searching energy-efficient, high-throughput, and low-latency mappings of deep learning loop nests onto a flexible deep learning accelerator based on reconfigurable interconnect called MAERI. Since the mapping search space is a high-dimensional optimization problem, we leverage deep-learning domain-specific and MAERI-specific heuristics to trim down the search space. Our analysis on realistic workloads shows that the no single mapping is best for entire DNN layers, and mRNA effectively searches and proposes mappings that maximize energy efficiency and/or minimize run time, which provides up to 26% and 67% lower runtime and up to 64% and 67% lower energy for convolutional and FC layers, respectively. We also demonstrate that the interconnect bandwidth plays a critical role in the mapping space search, and mRNA can be used to determine optimal mappings in bandwidth constrained designs. As future extensions, we plan to extend mRNA to support DNN layers with weight and input sparsity, which introduces irregular virtual neurons within MAERI and results in a more complicated mapping space than dense layers.

References

[1] Apple a12 processor. https://www.apple.com/iphone-xs/a12-bionic/, 2017.

[2] Xla. https://www.tensorflow.org/performance/xla/, 2017.

[3] MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects, author=Kwon, Hyoukjun and Samajdar, Ananda and Krishna, Tushar. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 461–475. ACM, 2018.

[4] MAERI: Enabling Rapid Design Space Exploration and Prototyping of DNN Accelerators. http://synergy.ece.gatech.edu/tools/maeri/, 2018.

[5] Mlperf. https://mlperf.org/, 2018.

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *ArXiv e-prints*, March 2016.

[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[8] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *International Symposium on Computer Architecture (ISCA)*, pages 662–673, 2018.

[9] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with

logical regions. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[10] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 38(3):247–257, 2010.

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

[12] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 609–622. IEEE Computer Society, 2014.

[13] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. Ramp: Resource-aware mapping for cgras. In *Design Automation Conference (DAC)*, pages 1271–1276. ACM, 2018.

[14] Jeremy Fowers et al. A configurable cloud-scale DNN processor for real-time AI. In *International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE Press, 2018.

[15] V. Govindaraju, C. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, 2012.

[16] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 152–159, 2017.

[17] M. Hamzeh, A. Shrivastava, and S. Vrudhula. Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In *ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10, 2013.

[18] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Epimap: Using epimorphism to map applications on cgras. In *Design Automation Conference (DAC)*, pages 1284–1291. ACM, 2012.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[20] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *International Conference on World Wide Web*, pages 173–182, 2017.

[21] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W Fletcher. UCNN: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 674–687. IEEE Press, 2018.

[22] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.

[23] Rie Johnson and Tong Zhang. Effective use of word order for text categorization with convolutional neural networks. In *Proceedings of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 103–112, 2015.

[24] Norman P Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems (NIPS)*, 2012.

[26] Hyoukjun Kwon, Michael Pellauer, and Tushar Krishna. Maestro: An analytic model for cost-benefit analysis of dataflows in dnn accelerators. *arXiv preprint arXiv:1805.02566*, 2018.

[27] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Rethinking nocs for spatial neural network accelerators. In *IEEE/ACM International Symposium on Networks-on-Chip*, page 19. ACM, 2017.

[28] HyoukJoong Lee, Kevin J Brown, Arvind K Sujeeth, Tiark Rompf, and Kunle Olukotun. Locality-aware mapping of nested parallel patterns on gpus. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 63–74. IEEE, 2014.

[29] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An instruction set architecture for neural networks. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 393–405. IEEE Press, 2016.

[30] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564. IEEE, 2017.

[31] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. In *International Symposium on Computer Architecture (ISCA)*, pages 416–429. ACM, 2017.

[32] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 495–506. ACM, 2013.

[33] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 17. IEEE Press, 2016.

[34] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2015.

[35] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations (ICLR)*, 2015.

[36] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[37] Z. Zhao, W. Sheng, N. Jing, W. He, and Z. Mao. Resource-saving compile flow for coarse-grained reconfigurable architectures. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2015.