# dMazeRunner: Executing Perfectly Nested Loops on Dataflow Accelerators

SHAIL DAVE, Arizona State University, US
YOUNGBIN KIM, Yonsei University, South Korea
SASIKANTH AVANCHA, Parallel Computing Lab, Intel Labs, India
KYOUNGWOO LEE, Yonsei University, South Korea
AVIRAL SHRIVASTAVA, Arizona State University, US

Dataflow accelerators feature simplicity, programmability, and energy-efficiency and are visualized as a promising architecture for accelerating perfectly nested loops that dominate several important applications, including image and media processing and deep learning. Although numerous accelerator designs are being proposed, how to discover the most efficient way to execute the perfectly nested loop of an application onto computational and memory resources of a given dataflow accelerator (*execution method*) remains an essential and yet unsolved challenge. In this paper, we propose *dMazeRunner* – to efficiently and accurately explore the vast space of the different ways to spatiotemporally execute a perfectly nested loop on dataflow accelerators (execution methods). The novelty of dMazeRunner framework is in: i) a holistic representation of the loop nests, that can succinctly capture the various execution methods, ii) accurate energy and performance models that explicitly capture the computation and communication patterns, data movement, and data buffering of the different execution methods, and iii) drastic pruning of the vast search space by discarding invalid solutions and the solutions that lead to the same cost. Our experiments on various convolution layers (perfectly nested loops) of popular deep learning applications demonstrate that the solutions discovered by dMazeRunner are on average 9.16× better in Energy-Delay-Product (EDP) and 5.83× better in execution time, as compared to prior approaches. With additional pruning heuristics, dMazeRunner reduces the search time from days to seconds with a mere 2.56% increase in EDP, as compared to the optimal solution.

CCS Concepts: • **Hardware** → **Hardware accelerators**; **Hardware-software codesign**; • **Computer systems organization** → *Reconfigurable computing*; • **Software and its engineering** → *Compilers*;

Additional Key Words and Phrases: Coarse-grained reconfigurable array, dataflow, deep neural networks, loop optimization, energy-efficiency, systolic arrays, mapping, analytical model, design space exploration

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2019.

Authors' addresses: S. Dave and A. Shrivastava, Compiler Microarchitecture Lab, School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ, US; emails: {Shail.Dave, aviral.shrivastava}@asu.edu; Y. Kim and K. Lee, Yonsei University, Seoul, South Korea; emails: {yb.kim, kyoungwoo.lee}@yonsei.ac.kr; S. Avancha, Parallel Computing Lab, Intel Labs, Bangalore, India; email: sasikanth.avancha@intel.com.

1539-9087/2019/10-ART70 $15.00
https://doi.org/10.1145/3358198

**ACM Reference format:**
Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. 2019. dMazeRunner: Executing Perfectly Nested Loops on Dataflow Accelerators. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 70 (October 2019), 27 pages.
https://doi.org/10.1145/3358198

## 1 INTRODUCTION

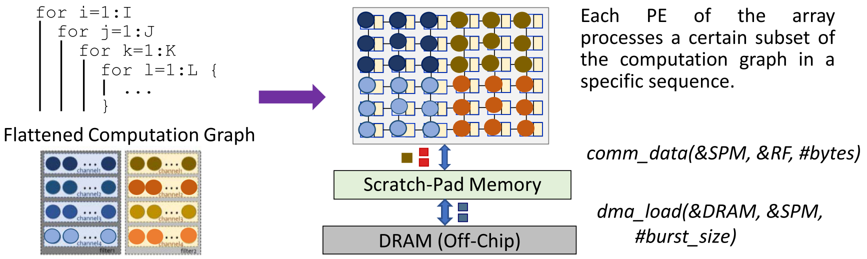
At the heart of several important-to-accelerate applications, e.g., multimedia, imaging, and deep learning are *perfectly nested loops*, which are often compute- and memory-intensive. A perfectly nested loop is a nested loop, where all the assignment instructions are inside the innermost loop. For example, the convolution kernel (that executes for the majority of execution time in ResNeXt [1]) is a 7-deep perfectly nested loop. Variations of dataflow accelerators, like *systolic arrays* (e.g., Tensor Processing Unit), *coarse-grained reconfigurable arrays (CGRAs)*, and spatial architectures are repeatedly being demonstrated as a promising accelerator for these power and performance-critical loops [2–10]. As shown in Figure 1, dataflow accelerators, in general, comprise an array of *processing elements aka PEs* (where PEs are function-units with little local control) and non-coherent *scratchpad memories (SPM)* that allow concurrent execution and explicit data management. While the simpler design makes the dataflow accelerator power-efficient, with adequate data prefetching and high data reuse at scratchpad and register file levels, PE array can be engaged continuously in useful computations, which results in high throughput and energy efficiency [3, 5].

However, how to discover the most efficient way to execute a perfectly nested loop of an application onto the computational and memory resources of a given dataflow accelerator (execution method) remains an essential and yet unsolved challenge. This is because, the joint search space of hardware design of the accelerator, combined with the ways to execute the loops both spatially and temporally on it, is vast. In other words, not only the architecture can be configured in many different ways, but for each of those configurations, the number of ways to answer questions like – how to divide the loop execution among PEs, which PEs processes what subset of the data and in which sequence, when to schedule the data movement between memory-levels of the accelerator (for data prefetching), and how much buffering to do in SPM – are numerous.

We refer to the different ways in which a perfectly nested loop can be executed on the dataflow accelerator as *execution methods*. When a programmer chooses a way of spatiotemporal execution of the loop-nest, that leads to a particular execution method. – Execution methods significantly impact the computation and communication patterns within the accelerator and therefore, the power and performance of the execution. – If they are not optimized/chosen well, acceleration benefits may even be negative!. In the absence of a systematic and explicit way to capture and explore vast design space, prior techniques have considered only certain execution methods (like row-stationary [11], output-stationary [12, 13] mechanisms for convolutions). Hence, they end up exploring only a tiny fraction of the space, during manual tuning [6] or randomization-based search [14, 15]. Furthermore, to meaningfully search the vast design space, an accurate analytical model (which can determine the goodness of an execution method) is required. Although [16, 17, 18] developed analytical models, they either lack estimation of the execution time or energy consumption and are specific to DNNs.

In this paper, we propose **dMazeRunner** (pronounced as *the maze runner*) – a framework to efficiently and accurately explore vast design space of different execution methods to execute perfectly nested loops on dataflow accelerators. dMazeRunner includes:

**Holistic representation that captures the vast space of execution methods:** The dataflow execution on the accelerator takes place by executing loop iterations spatially onto PEs and by



```
for i=1:I
  for j=1:J
    for k=1:K
      for l=1:L {
        ...
      }
```

Flattened Computation Graph
Each PE of the array processes a certain subset of the computation graph in a specific sequence.
comm_data(&SPM, &RF, #bytes)
Scratch-Pad Memory
dma_load(&DRAM, &SPM, #burst_size)
DRAM (Off-Chip)

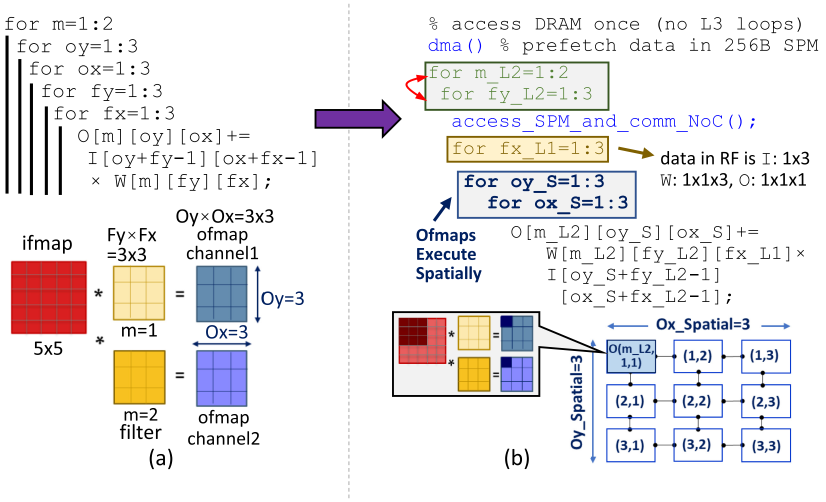
Fig. 1. Programming the dataflow accelerators requires explicit management of computational, memory, and communication resources.

managing the data accesses from RF, SPM, and DRAM. So, dMazeRunner uses a representation which features an explicit tiling of the loop-nest at these four levels. For example, explicit tiling of a 7-deep nested loop into a 28-deep nest ensures that all variations of spatial execution and the data reuse in RFs and SPM are succinctly captured. With the loop iteration counts (tiling factors) and ordering of the loops as configuration parameters, the proposed representation captures the vast space of execution methods.

**Drastic pruning of the vast search space:** The explicit representation of the vast space enables dMazeRunner to systematically explore and prune the search space. dMazeRunner analyzes the loop-nest and constructs a list of only those loop-orderings that feature unique reuse factors of the data operands (prunes to 15 schedules from 7! = 5040 orderings for a seven-deep loop-nest of convolution kernels). Additionally, dMazeRunner considers only valid loop tiling and PE array partitioning options. To cut down the exploration time, dMazeRunner caches the commonly invoked routines and explores the search space with multi-threading. To reduce the search space further, dMazeRunner can employ pruning heuristics (sub-optimal) to attain an efficient solution promptly. For example, pruning heuristics only consider execution methods that: i) achieve high utilization of architectural resources, ii) do not access non-contiguous data from DRAM, iii) do not require inter-PE communication, and iv) maximize the reuse of data operands. dMazeRunner does not preclude experienced programmers from performing directed exploration of the search and design space, but rather enable a rapid and systematic search (within succinctly captured vast search space) such that even domain non-experts can achieve highly efficient execution on dataflow accelerators.

**Analytical modeling of execution methods:** dMazeRunner analyzes any given execution method for a perfectly nested loop and estimates the energy consumption and execution time. dMazeRunner explicitly models the computation and communication patterns of execution, including determining the various data reuse factors, DMA invocations and burst size for managing non-contiguous data in SPM, data buffering options, miss penalty, data distribution through network-on-chip (NoC), and inter/intra-PE-group communication (for reduction operations). dMazeRunner takes architecture specification of the dataflow accelerator as an input, which can be varied in terms of a number and organization of PEs, the memory sizes and configurations, NoC configuration, and DMA model.

Note that we use convolution layers from deep neural network (DNN) models to explain the background and examples and for demonstrating the search space and design space exploration capabilities of dMazeRunner. This is because, convolution layers in DNN models feature 7-deep loop-nest (dense than matrix multiplication or other applications), exhibiting various ways of data reuse and spatial execution. They are widely used in deep learning and media processing applications [1, 2, 19–23]. However, our approach is more general and can optimize the execution of any



```
for m=1:2
 for oy=1:3
  for ox=1:3
   for fy=1:3
    for fx=1:3
      O[m][oy][ox]+=
       I[oy+fy-1][ox+fx-1]
       × W[m][fy][fx];
```

```
% access DRAM once (no L3 loops)
dma() % prefetch data in 256B SPM
for m_L2=1:2
 for fy_L2=1:3
  access_SPM_and_comm_NoC();
  for fx_L1=1:3
   for oy_S=1:3
     for ox_S=1:3
      O[m_L2][oy_S][ox_S]+=
       W[m_L2][fy_L2][fx_L1]×
       I[oy_S+fy_L2-1]
        [ox_S+fx_L2-1];
```

Fig. 2. (a) Convolution of 5×5 ifmap with 3×3 weights of 2 filters. (b) An execution method, which executes 3×3 ofmap on different PEs of the dataflow accelerator.

perfectly nested loop (featuring direct memory accesses and statically known loop bounds) on a dataflow accelerator.

We validate the dataflow execution model of dMazeRunner against evaluations of [3, 18, 24] for the same execution methods. The energy consumption and PE utilization achieved by our model closely matches the model [24] (energy differs by 4.19%). Moreover, when validated against Eyeriss architecture [3], estimation of the execution time differs by 11%. Owing to exhaustive and superior search space exploration capabilities, for various convolution layers from popular ResNet and ResNeXt [1, 20] applications, dMazeRunner finds execution methods that outperform the prior techniques and reduces total EDP by 9.16× on average and total execution cycles by 5.83×. Furthermore, after employing pruning heuristics, dMazeRunner cuts down the exploration space by over 9000× (reducing the optimization search time from days to few seconds), with only 2.56% increase in EDP, as compared to the optimal solution.

## 2 SPATIOTEMPORAL EXECUTION OF LOOPS ON DATAFLOW ACCELERATORS

The efficiency of executing a perfectly nested loop onto a dataflow accelerator depends on the execution method which defines the spatiotemporal organization of loop iterations. If all loop iterations are processed simultaneously on different PEs, then execution would finish in one shot. However, due to a limited number of PEs, only some loops are (partially) executed in space, and remaining loops iterate temporally on each PE. For example, consider the loop-nest of Figure 2(a), which is a simplified convolution kernel. It shows that a convolution of a 5×5 *input feature map (ifmap)* with 3×3 weights of two filters yields two output channels of 3×3 *output feature map (ofmap)*. All data elements are of 16 bits. Now, Figure 2(b) shows one execution method to map the nest of Figure 2(a) onto a sample dataflow accelerator consisting 3×3 PEs, where each PE accesses own 16B RF and a 256B shared SPM. For example, executing the loop with an *index variable (IV)* `ox` in space requires a row of 3 PEs in the accelerator. Similarly, spatially executing both the loops with IVs `ox` and `oy` requires 3×3 PEs. Here, each PE computes a unique ofmap value `O(m_L2,oy_S,ox_S)` while temporally executing loops with IVs *m*, `fx`, and `fy`. PE(1,1) corresponds to `oy_S`=1 and `ox_S`=1. So, PE(1,1) processes `O(m_L2,1,1)` and requires ifmaps I(1,1)–I(3,3) and all the weights W(1,1,1)–W(2,3,3). In contrast, if some other execution method corresponds to executing loops with IVs fx and fy in space, then each PE will maintain different weights and will generate a partial outcome. Thus, **selecting which loops are (completely or in part) executed in the space determines what subset of the data gets processed by each PE.**

**The organization of the loops that execute temporally on each PE determines the exact sequence of processing the data** and thus, significantly impacts the data reuse and data management of RFs and SPM. For example, for the execution method of Figure 2(b), loops with IVs *m*, fy, and fx execute temporally. The loop corresponding to the columns of the filters (fx) executes at level 1. This implies that the data corresponding to the loop with IV fx_L1 is buffered into RFs (L1 memory) of PEs. The execution method allocated data into RFs at maximum capacity (3 elements for I and W, 1 element for O i.e., 7 elements or 14 bytes in 16-byte RFs). Thus, each PE executes 3 times (fx_L1=1:3) and processes data from the registers. Now, when the remaining loops execute (a total of 6 iterations of L2 loops with IVs fy_L2 and m_L2), new data is accessed from the SPM (L2 memory) and communicated to PEs via NoC. Since the operand O is invariant of fy_L2, it gets used thrice from RFs of PEs. Thus, both the ifmaps and weights are loaded from SPM 2×3 = 6 times, while ofmap is reused and written to SPM just twice. Now, after interchanging both the L2 loops, the loop with IV m_L2 becomes innermost. Hence, with I being invariant of m_L2, ifmap gets reused.

Note that the execution method of Figure 2(b) shows just one way of spatiotemporal execution and many such variations are possible. However, when execution methods are not explicitly modeled (e.g., in the code of Figure 2 a), a specific execution sequence is implicit, and it is impracticable to capture and explore the variations in both the spatial execution and data reuse in memory hierarchy.

## 3 RELATED WORK

**Dataflow accelerator architectures:** Several dataflow accelerator designs are proposed recently [2, 3, 5, 25]. Google TPU [2] is a systolic-array accelerator for DNNs and LSTMs (long short-term memory). Chen et al. [3, 11] proposed Eyeriss architecture that efficiently executes their novel row stationary dataflow mechanism. Cong et al. [26] used a polyhedral based analysis to generate high-performance systolic array architectures for executing loops on FPGAs. HyPar architecture [27] is an array of hybrid memory cube based accelerators for training DNNs. Lu et al. [5] considered various dataflow mechanisms to execute convolutions and proposed a dataflow accelerator architecture which can execute either of them.

**Compilation techniques for loop optimizations:** Although techniques of loop tiling and permutation are well studied over the past few decades, they are either agnostic to hardware features or primarily researched for off-the-shelf processors [28–31]. Moreover, their cost functions are often limited to the memory subsystem of a processor with an objective to optimize the data allocation in the on-chip memory. However, minimizing DRAM accesses is not sufficient to achieve efficient mappings for dataflow accelerators, since other factors like efficient interleaving of computation with communication, efficient reuse of different operands, and higher resource utilization significantly contribute to the net acceleration. In fact, due to diverse architectural features (pipelined PEs, data buffering options, NoC configurations, memory sizes, and memory configurations), complete modeling and optimization for the entire accelerator system are required. Furthermore, these loop optimization techniques may require drastic pruning for exploring the optimal execution method. For example, loop optimization techniques of [29, 32] suffer from the vast space of loop-orderings, since up to 7! = 5040 orderings (per tiling configuration) need to be explored for a 7-deep loop-nest. Besides, an alternative to MIMD-style dataflow execution is software pipelining the loops; loop operations of the same or consecutive iterations concurrently execute on PEs of a CGRA [33, 34]. Such an approach is beneficial to accelerating non-vectorizable loops through instruction-level parallelism. However, these mapping techniques were primarily evaluated for kernels with relatively small computational or memory requirements and on considerably smaller PE arrays (16–64 PEs) [33, 35, 36]. In contrast, high-performance demanding kernels of gemm, convolutions, logistic regressions, etc. exhibit abundant data- and thread-level parallelism and can be efficiently accelerated on the designs with larger arrays of PEs (e.g., from 256 to 65,536) featuring larger RFs.

**Explicit modeling of all execution methods:** For compute- and memory-intensive loop-nests, numerous execution methods exist for configuring tiling and ordering of the loops for their spatial execution and for accessing the data from RFs, SPM, and DRAM. In the absence of a system to explicitly and succinctly capture the vast space of execution methods, the programmers and architects considered specific execution methods. For example, [17, 37] tiled loop-nest once (transformed a 7-deep nest to 14-deep), which specified how accelerator accesses DRAM and buffer the data in SPM. However, they lacked tiling the loops further to explicitly model the spatial execution and RF accesses. This scenario is similar to the code of Figure 2(a), which implicitly assumed a sequence and offered no insight about variations in the data loaded from SPM to RF and how differently PEs can process data. Similarly, [12, 38] executed loops corresponding to ofmaps in the space, missing out exploring many execution methods. [12, 38] maximized the psum reuse, and [3] maximized weight reuse in RF and psum reuse in SPM and did not explore other execution methods. Likewise, [15, 37, 39, 40] considered a batch size of $N = 1$ images, missing the opportunities for weight reuse. Thus, prior techniques organized the loops in certain ways and *without explicit modeling of the complete spatiotemporal execution*, they lacked information about different execution methods. We demonstrate later that without a systematic approach (like the representation used by dMazeRunner) that captures vast space of the execution methods, information available about the entire space is not comprehensive. Hence, the programmer/optimizer ends up with an inferior solution.

**Pruning the search space:** The space of execution methods is vast because, total options for multi-level tiling of the loop-nest range from several hundred to thousands [39] and for each tiling configuration, loops are reordered in numerous ways. For example, we can organize a 7-deep loop-nest of convolution into $7! = 5040$ ways [37]. Collectively, this requires a vast space to explore (billions of execution methods!), and it has been infeasible to perform a brute-force search for the optimal execution method. Therefore, prior techniques *heuristically* reduced the search. For example, [3, 12] offered specific ways of spatiotemporal execution of convolutions, which are not always very efficient. In exploring various tiling configurations, [37] *fixed the order of specific loops*, to cut down the orderings of 6 loops from $6! = 720$ to 180. Likewise, [18, 39] *heuristically* reduced the space by limiting the options of tiling the loops. During FPGA design space exploration (DSE), [40] fixed the order of innermost loops (impacts data reuse) to simplify HLS code generation, and [15, 40] fixed a choice about which loops are spatially executed (impacts PE utilization). Chen et al. [14] developed a machine learning algorithm, which uses *simulated annealing* to predict an execution method based on the prior execution traces. Similarly, [15] employed a *genetic algorithm* based optimizer. However, for these techniques, without effective pruning, the search space remained vast. Therefore, when prior heuristics targeted only a tiny fraction of different execution methods, the obtained solution is not necessarily optimal or even close-to-optimal.

**Analytical modeling of dataflow execution:** For mapping perfectly nested loops onto dataflow accelerators and for DSE, it is necessary to determine the effectiveness of an execution method statically. Since dataflow accelerators exhibit simple design and are explicitly managed, few works recently developed analytical models to either estimate energy consumption or execution time [16–18] for DNNs. Table 1 lists the various features of such tools and their limitations. For example, MAESTRO [41] provides an analytical model for DNNs and estimates the efficiency

Table 1. Analytical Models for Design Space Exploration of Dataflow Execution

| Features | SCALE-Sim [16] | MAESTRO [41] | Yang et al. [24] | dMazeRunner |
|---|---|---|---|---|
| Programmer intervention is not required | ✓ | ✗ | ✓ | ✓ |
| Availability of an auto-optimizer | ✗ | ✗ | ✓ | ✓ |
| Availability of the energy model | ✗ | ✓ | ✓ | ✓ |
| Availability of the execution time estimation | ✓ | ✓ | ✗ | ✓ |
| Models miss penalty (for data communication latency) | ✗ | ✓ | ✗ | ✓ |
| Models stall cycles for reduction operation | ✗ | ✓ | ✗ | ✓ |
| Model is applicable to applications other than DNNs | ✗ | ✗ | ✗ | ✓ |
| Integrated support for common ML application libraries | ✗ | ✗ | ✗ | ✓ |

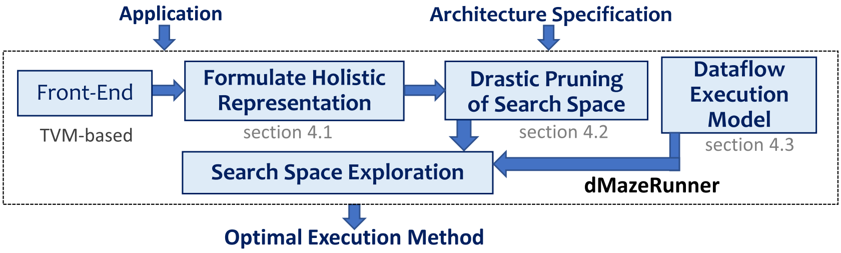


Fig. 3. Overview of dMazeRunner framework for application mapping onto dataflow accelerators.

of an execution method. However, the user needs to understand the proposed directives and write them in MAESTRO DSL, where chosen parameters for the directives can significantly impact the spatiotemporal execution. Yang et al. [18] proposed an energy model [24] for dataflow execution of DNNs and LSTMs, but it lacks estimation of the execution time. Likewise, [7, 17] proposed performance models with an assumption that PEs are always engaged in performing operations and never stall. Thus, prior analytical models either lack estimation of energy consumption or execution time, or do not accurately model data reuse or miss penalty, or lack auto-optimizer. Moreover, these models are specific to DNNs (i.e., may not be capable of analyzing nested loops from various applications). Furthermore, they require the user to specify DNN layer parameters as inputs and do not provide integrated support for common ML/AI application libraries like TensorFlow, MXNet, or PyTorch.

## 4 DMAZERUNNER

To efficiently map perfectly nested loops onto programmable dataflow accelerators, we propose dMazeRunner as a comprehensive solution. Figure 3 shows dMazeRunner framework. Its front-end parses the application and extracts target loop-nest. After analyzing the loop-nest, dMazeRunner formulates a holistic representation which features explicitly tiled loops for spatial execution as well as for accessing data from RFs, SPM, and DRAM. Various configurations of this representation capture the vast space of execution methods.

To formulate the space of execution methods, dMazeRunner analyzes data access patterns of the loop-nest and constructs a *list of only those loop orderings that correspond to unique reuse factors of data operands*. It considers only those *tiling factors* which are valid when subjected to constraints for loop functionality and capacity of architectural resources. Thus, by discarding invalid and redundant solutions, dMazeRunner prunes the search space so drastically that enables a brute-force exploration of the optimal solution. Furthermore, to achieve close-to-optimal solutions in second(s), dMazeRunner reduces the space further with heuristics. For example, it considers only those methods that maximize data reuse, do not require inter-PE communication, minimize DRAM accesses for non-contiguous data, and highly utilize architectural resources.

To determine the goodness of an execution method statically, dMazeRunner explicitly models computation and communication costs for various architectural features and estimates the execution time and energy consumption. From an input loop-nest, the model analyzes indexing expressions and data dependencies of operands. Then, the model determines various data reuse factors, DMA invocations and burst sizes for managing non-contiguous data in SPM, miss penalty, data communication through NoC, and any stall cycles inter/intra-PE-group communication (for reduction operations).

**Framework implementation:** dMazeRunner framework features analysis, transformations, and optimizations for dataflow execution of loops. Front-end of the framework leverages TVM environment [14] to support various applications and multiple ML libraries such as MXNet, keras, and TensorFlow. Using the TVM environment, dMazeRunner achieved execution method can be transformed into LLVM IR [42] for code generation. Moreover, for a rapid design space exploration on modern multi-core platforms, our framework implementation leverages the hardware features like caching of the commonly invoked analysis routines and multi-threading. The framework is available at https://github.com/cmlasu/dMazeRunner.

## 4.1 Holistic Representation to Capture Vast Space of Execution Methods

Execution on the dataflow accelerators takes place by means of executing the loop iterations onto the PE array both spatially and temporally. To determine spatial execution onto PEs and the data accessed from RFs, SPM, and DRAM, we explicitly tile each loop of the loop-nest at these four levels. Figure 4(b) shows the proposed representation, which is obtained after transforming the algorithm of Figure 4(a). Thus, dMazeRunner transforms a 7-deep nested loop into a 28-deep nested loop. In this explicitly tiled form, the configurable parameters are—loop iteration counts (tiling factors like *N_SPATIAL*, *N_RF*) and ordering of the loops at any of 4 levels.

This representation can be configured to represent various execution methods. For example, to achieve the method of Figure 2(b), we first configure the 7 innermost loops that correspond to spatial execution. The innermost two loops (`ox` and `oy`) that have tiling factors greater than 1 (`Ox_SPATIAL = 3`, `Oy_SPATIAL=3`) determine how PEs are grouped in a 2D array. For example, Figure 5 shows tiling for spatial execution of three loops. Here, unrolling the third tiled loop (`M_SPATIAL=2`) for spatial execution results in two groups of 3×3 PEs. In fact, if the hardware features interconnections for 3D array (e.g., cubic or vertically-stacked 2D array), then such tiling for spatially executing more than two loops can be translated into mapping onto a 3D array.

The seven loops at levels L1, L2, and L3 execute temporally on each PE and are configured to specify the accesses to RF, SPM, and DRAM. Here, tiling factors (e.g., `N_SPM=2`) impact the size of the data accessed from L1/L2/L3 memory (Section 4.3.1 provides the exact calculation), and ordering of the loop determines the schedule of data movement i.e., data reuse/eviction. In the proposed representation, since each loop of the input nest is *explicitly* modeled for spatial execution and for accessing data from L1/L2/L3 memory, it allows capturing the vast space of execution methods.

## 4.2 Drastically Pruning the Vast Search Space

*4.2.1 Determining Loop Orderings for Unique Data Movement Costs.* While tiling factors for L1, L2, and L3 loops determine the size of the data accessed from RF, SPM, and DRAM, the orderings of these loops determine the data reuse and scheduling of the data movement. In a loop nest, data

```
for n=1:N                % batch size
 for m=1:M               % filters
  for c=1:C              % channels
   for oy=1:Oy           % ofmap rows
    for ox=1:Ox          % ofmap cols
     for fy=1:Fy         % filter height
      for fx=1:Fx        % filter width
       O[n][m][oy][ox] +=
        I[n][c][oy+fy-1][ox+fx-1]
        × W[m][c][fy][fx];
```

(a)

```
for n_L3 = 1:N_DRAM
 for m_L3 = 1:M_DRAM
  for c_L3 = 1:C_DRAM
   for oy_L3 = 1:Oy_DRAM
    for ox_L3 = 1:Ox_DRAM
     for fy_L3 = 1:Fy_DRAM
      for fx_L3 = 1:Fx_DRAM
       {
        dma( );
        for n_L2 = 1:N_SPM
         for m_L2 = 1:M_SPM
          for c_L2 = 1:C_SPM
           for oy_L2 = 1:Oy_SPM
            for ox_L2 = 1:Ox_SPM
             for fy_L2 = 1:Fy_SPM
              for fx_L2 = 1:Fx_SPM
               {
                access_SPM_and_comm_NoC();
                for n_L1 = 1:N_RF
                 for m_L1 = 1:M_RF
                  for c_L1 = 1:C_RF
                   for oy_L1 = 1:Oy_RF
                    for ox_L1 = 1:Ox_RF
                     for fy_L1 = 1:Fy_RF
                      for fx_L1 = 1:Fx_RF
                       {
                        for n_S = 1:N_SPATIAL
                         for m_S = 1:M_SPATIAL
                          for c_S = 1:C_SPATIAL
                           for oy_S = 1:Oy_SPATIAL
                            for ox_S = 1:Ox_SPATIAL
                             for fy_S = 1:Fy_SPATIAL
                              for fx_S = 1:Fx_SPATIAL
                                O[][][][] +=
                                 I[][][][] ×
                                 W[][][][];
                       }
               }
       }
```

- Iteration counts (tiling factors) and orderings for L3 loops determine data communicated to (reused in) SPM.
- Iterations of L2 loops affect SPM accesses and the cost of data communication to RF via NOC.
- Iterations of L1 loops indicate data accessed by a PE from RF.
- Inner loops are unrolled in space; Spatial execution determines the subset of the data processed by each PE.

(b)

Fig. 4. Explicitly tiled representation that models the vast space of different execution methods.

```
% access DRAM once
dma() % prefetch in SPM
for fy_L2=1:3
 access_SPM_comm_NoC();
 for fx_L1=1:3
  for m_S=1:2
   for oy_S=1:3
    for ox_S=1:3
     O[m_S][oy_S][ox_S]+=
      W[m_S][fy_L2][fx_L1]×
      I[oy_S+fy_L2-1]
       [ox_S+fx_L1-1];
```

```
% loops for temporal execution at L1,L2,L3
{
 for n_S = 1:N_SPATIAL=1
  for c_S = 1:C_SPATIAL=1
   for fy_S = 1:Fy_SPATIAL=1
    for fx_S = 1:Fx_SPATIAL=1
     for m_S = 1:M_SPATIAL=2
      for oy_S = 1:Oy_SPATIAL=3
       for ox_S = 1:Ox_SPATIAL=3
        O[][][][] +=
         I[][][][] ×
         W[][][][];
}
```

Configuring tiling for spatially executing three loops.

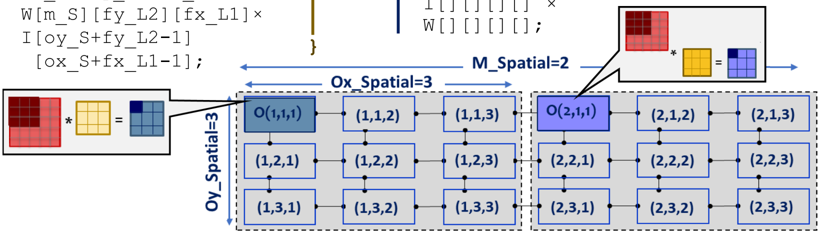
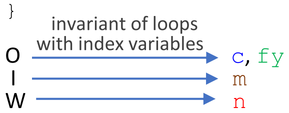


Fig. 5. Configuring the representation of Figure 4(b) for spatial execution of three loops, which results in two PE-groups in the accelerator.

```
for n = 1:N=2
 for m = 1:M=8
  for c = 1:C=4
   for fy = 1:Fy=3 {
     comm_data();
     O[n][m]+=
       I[n][c][fy]×
       W[m][c][fy]
   }
```



**Unique Reuse Factors for Data Operands**

| Loop Order | I | W | O |
|---|---|---|---|
| {..., n} | – | N = 2 | – |
| {..., m} | M = 8 | – | – |
| {..., m, c} | – | – | C = 4 |
| {..., m, fy} | – | – | Fy = 3 |
| {..., fy, c} | – | – | C×Fy = 4×3 |

Fig. 6. Determining all orderings of loops that feature unique data reuse factors. Achieved orderings are five among a total of 4! = 24 orderings. Dash symbol indicates a use factor of 1 for an operand (i.e., no reuse).

operands (tensors) are often invariant of specific loops and can be reused [29]. Therefore, for a given loop-nest, it is possible to create a list of all those *loop-orderings (schedules)* that feature unique reuse of operands, and the optimizer needs to target just those orderings. For example, we demonstrate that out of 7! = 5040 orderings to organize 7-deep loop-nest of convolution, loop-orderings featuring unique reuse factors are just up to 15. Such reduction stems from the fact that for execution of tiled L2/L3 loops, memory management ensures the availability of the data blocks prior to the execution, and reuse factors of operands (data blocks) get limited, as compared to numerous hit/miss occurrences possible (at cache-line granularity) in a cache-based memory hierarchy.

Figure 6 depicts a 4-deep loop-nest along with information about each operand being invariant of certain loops. To explain the impact of orderings, in this example, we assume that the current memory level (e.g., RF) can accommodate 3 data elements. Thus, during each loop iteration (total 192), the data corresponding to each operand can be accessed from lower memory (e.g., SPM) and brought to the current memory level. In other words, for a given ordering, for each operand, the function *comm_data*() may (not) execute in every loop iteration. Figure 6 also tabulates different orderings that feature unique reuse factors. Loop IVs are in lower-case, and *trip-counts (TCs)* are in upper-case. For each schedule, a listing of loop IVs from the right- to left-hand-side indicate the order of innermost to outermost loops. For example, the first ordering indicates that loop with IV $n$ is the innermost, and "..." indicates that the ordering of outer 3 loops does not matter for this schedule. So, selecting any one ordering among 3! combinations yields the same reuse.

To generate the schedules (Algorithm 2), dMazeRunner iterates over each operand and constructs the loop-orderings for which the operand is invariant of inner loops. For example, W is invariant of n, and the first loop-ordering is the only schedule where W is reused for N=2 iterations. Thus, out of 192 iterations, W is accessed from memory only 192/2 = 96 times. However, since I and O are indexed through n, they are communicated from lower memory during all 192 iterations (for a given ordering, Algorithm 1 determines such reuse factors for operands). Similarly, I gets reused only in the second ordering. Now, O is invariant of two IVs c and fy (total_independent_IVs=2). So, more than one orderings feature unique reuse of O (generated by lines 5–15 of Algorithm 2). Two possible orderings ($3^{rd}$ and $4^{th}$) are where O is reused only in the innermost loop with IV as either c or fy. Similarly, O is reused in both the inner loops when IVs for inner loops are permutations of c and fy. Here, both the permutations ('c', 'fy') or ('fy', 'c') yield the same reuse factors (1 for I and W and 12 for O). So, we consider any 1 permutation (line 16 in Algorithm 2 prunes another), which is $5^{th}$ ordering. Thus, dMazeRunner prunes 4! = 24 orderings to just 5. Similarly, for convolution of Figure 4(a), dMazeRunner prunes 7! = 5040 orderings to 15 orderings that feature unique reuse, which are listed in Table 2. For given tiling factors of an execution method,

Table 2. Unique Data Reuse Factors for Accessing Lower Memory

| **Schedule** | **Ifmap** | **Weights** | **Ofmap** |
|---|---|---|---|
| { . . . , **m**} | M | 1 | 1 |
| { . . . , m, **ox**} | 1 | Ox | 1 |
| { . . . , m, **oy**} | 1 | Oy | 1 |
| { . . . , m, **n**} | 1 | N | 1 |
| { . . . , m, **oy, ox**} | 1 | Oy × Ox | 1 |
| { . . . , m, **n, ox**} | 1 | N × Ox | 1 |
| { . . . , m, **n, oy**} | 1 | N × Oy | 1 |
| { . . . , **n, oy, ox**} | 1 | N × Oy × Ox | 1 |
| { . . . , m, **fx**} | 1 | 1 | Fx |
| { . . . , m, **fy**} | 1 | 1 | Fy |
| { . . . , m, **c**} | 1 | 1 | C |
| { . . . , m, **fy, fx**} | 1 | 1 | Fy × Fx |
| { . . . , m, **c, fx**} | 1 | 1 | C × Fx |
| { . . . , m, **c, fy**} | 1 | 1 | C × Fy |
| { . . . , **c, fy, fx**} | 1 | 1 | C × Fy × Fx |

**ALGORITHM 1:** *Determine_Data_Reuse*(Input *loop_ordering*, Input *level*, Input *operand_list*, Output *reuse_vector*)

```
foreach operand op in operand_list do
    operand_reuse_factor = 1;
    list_op_dependent_IVs = get_op_dependencies(op);
    foreach iv in reversed(list(loop_ordering)) do
        tc = get_TripCounts(iv, level);
        if (tc == 1) then
            continue;
        else if (iv is not in list_op_dependent_IVs) then
            operand_reuse_factor *= tc;
        else
            break;
    reuse_vector[op] = operand_reuse_factor
return reuse_vector
```

collective orderings (of L2 and L3 loops) to reuse the data while accessing SPM and DRAM are up to 15×15 instead of 5040×5040. Note that the list of orderings (e.g., ones in Table 2) are determined statically once, before the exploration and evaluation of execution methods begin. Furthermore, during exploration of execution methods, for a given set of tiling factors, it is possible that one or more loops iterate(s) just once (e.g., M_SPM=1). In such a scenario, among these 15 orderings, several orderings feature the same reuse factors. In other words, unique reuse factors reduce from 15 orderings. Thus, during exploration, for each set of tiling factors, dMazeRunner dynamically prunes the list of 15 orderings (of Table 2) further.

dMazeRunner constructs the list of orderings depending on the operand being invariant of the loops, which is determined by analyzing the indexing expressions of the operand (e.g., I is invariant of IV m). Therefore, the proposed pruning technique is applicable to direct memory access patterns (including affine accesses), which are commonly found in many applications. Note that

**ALGORITHM 2:** *Generate_Loop_Orderings*(Input *operand_list*, Output *pruned_orderings*)

```
foreach operand op in operand_list do
    list_op_independent_IV = get_op_dependencies(op);
    total_independent_IVs = len(list_op_independent_IV);
    list_orderings = null; iter = 1;
    while $iter \le total\_independent\_IVs$ do
        list_IVs = null;temp_list = get_combinations_IVs(list_op_independent_IV, iter);
        foreach item in temp_list) do
            list_permutations = get_all_permutations(item);
            list_IVs.append(list_permutations);
        list_IVs.remove_duplicate_items();
        foreach item in list_IVs) do
            temp_ordering = prepend_dependent_IVs(item, list_op_dependent_IV);
            order = prepend_missing_IVs_in_random_order (temp_ordering,
              list_op_independent_IVs);
            list_orderings.append(order);
        iter++;
    pruned_orderings= prune_orderings_same_reuse(list_orderings);
return pruned_orderings
```

in determining orderings, a loop interchange is considered only when it is a legal transformation. The legality can be determined by analyzing distance- and dependence-vectors for the loops [28].

*4.2.2 Determining Valid Tiling Options.* After multi-level tiling of a loop, TCs of the tiled loops can be of any integer value. For example, consider a loop that iterates $N = 8$ times. After tiling it into 4-levels, TCs of the tiled loops are $N\_SPATIAL$, $N\_RF$, $N\_SPM$, and $N\_DRAM$, which are optimization parameters. When off-the-shelf optimizers (constraint-solvers for non-linear programming that use simulated-annealing, newton's method, etc.) are employed [15, 43], in each step, they randomly select the parameter values from all possible combinations ($8^4$). For large-scale optimization problems, since the valid methods are very few (e.g., 20 out of 4096 in this example), their majority of the search time is often spent on discarding invalid solutions. However, dMazeRunner employs a constraint-driven pruning of the space before beginning the exploration and analytical evaluation of execution methods, by considering only valid tiling options (e.g., 20 instead of $8^4$). For example, it ensures that for tiling of a loop into four loops, the total iterations executed by the tiled loops match the functionality of the loop-nest, i.e.,

$$cons : N\_SPATIAL \cdot N\_RF \cdot N\_SPM \cdot N\_DRAM = N$$

In general, for any loop index-variable `iv`,

$$TC[base][iv] = TC[SPATIAL][iv] \cdot TC[RF][iv] \cdot TC[SPM][iv] \cdot TC[DRAM][iv]$$

dMazeRunner also ensures that the pruning is subjected to constraints from architecture resources (PEs, RF, and SPM). For example, data to be allocated by an execution method (section 4.3.1) must fit into RF of a PE and in multi-buffer SPM, i.e.,

$$cons : \sum_{op=1}^{total\_Operands} data\_alloc[RF][op] \le RF\_size$$

$$cons: \sum_{op=1}^{total\_Operands} data\_alloc[SPM][op] \leq SPM\_size$$

$$cons: \prod_{i=1}^{total\_IVs} TC[SPATIAL][IVi] \leq Total\_PEs$$

For example, when RF tiling factors <N_RF, M_RF, C_RF, Oy_RF, Ox_RF, Fy_RF, Fx_RF>are selected as <1,1,1,1,1,1,3>, allocated registers for weights are `data_alloc[RF][W]` = M_RF×C_RF×Fy_RF×Fx_RF = 3. Total allocated registers are 3+3+1 = 7 (for I, W, and O), and this is a valid method for an 8-element RF (example of Figure 2). However, a solution with RF tiling factors <2,1,1,1,1,1,3> is invalid and not considered for the exploration, since it allocates 6+3+2 = 11 elements. Thus, the constraints discard invalid tiling options and with eliminating numerous orderings that feature the same costs, dMazeRunner drastically prunes the space. Hence, it enables a brute-force exploration of execution methods, achieving the optimal solution.

*4.2.3 Pruning the Space with Heuristics to Rapidly Achieve Close-to-Optimal Solution.* Depending on the depth and iteration counts of the loops in the application, the exhaustive exploration may take even several hours. One strategy can be to pre-compile the application for common target architectures, where the optimal execution method is explored just once. However, to allow re-compiling applications by users and rapid design space explorations, the optimizer should be able to generate a highly efficient solution promptly. So, dMazeRunner embeds a pruning heuristic that achieves close-to-optimal solutions in second(s) through the following strategies:

**OPT 1) Targeting execution methods featuring high resource utilization:** dMazeRunner explores only those tiling factors that highly utilize (e.g., 60%) RFs, SPM, and PEs. High utilization improves data reuse and reduces DRAM accesses. Note that very high utilization does not guarantee an optimal solution, as it may not effectively interleave computation and communication cycles.

**OPT 2) Discard execution methods requiring several memory accesses of non-contiguous data:** Some IVs of loops correspond to a minor dimension of tensors (`fy` and `fx` for $W$[m][c][fy][fx]). For such IVs, when tiling factors of L3 loops (i.e., `Fy_DRAM`) are greater than 1, it requires many DMA invocations with small burst-sizes. Thus, it results in higher DMA cycles and may introduce the miss penalty for SPM management. So, dMazeRunner discards such execution methods which are susceptible to higher execution time.

**OPT 3) Discard execution methods that require inter-PE communication:** Often a *read + write (r+w)* operand ($O$) is an invariant of few IVs (`c`, `fy`, and `fx`). If loops corresponding to these IVs execute spatially, it requires inter-PE communication (for reduction), which may introduce stall cycles and often costs higher energy. Therefore, to avoid inter-PE communication, dMazeRunner decides not to execute such loops in space. This strategy discards several dataflow mechanisms (e.g., weight-stationary, row-stationary).

**OPT 4) Targeting execution methods that maximize the reuse of operands:** Although dMaze-Runner determines all loop-orderings featuring unique reuse factors, space can be pruned to few orderings that maximize the data reuse. For example, in Table 2, only schedules #8 and #15 maximize the reuse of weights and ofmap respectively. Thus, schedules #2–#7 and #9–#14 are discarded.

**OPT 5) Leveraging hardware features of compilation platform:** Implementation of dMazeRunner framework integrates - (i) caching of the frequently used analysis routines and commonly referenced hash tables (e.g., loop orderings), and (ii) concurrently exploring various execution methods and evaluating their efficacy with multi-threading. Thus, on modern multi-core processors, the exploration time is significantly reduced. Note that OPT4 and OPT5 do not impact optimality and can be used for an exhaustive search.

### 4.3 Dataflow Execution Model

*4.3.1 Determining Data Allocation.* Table 3 lists the notations used in describing analytical model, along with the interpretation. For the given tiling factors of an execution method, the data to be allocated in RF of a PE, in SPM, and the data communicated to the PE array is determined as:

$$data\_alloc[option][op] = evaluate\_index\_expr(op, effective\_TC)$$

where, for each $iv$ in the list `IV`,

$$effective\_TC[iv] = \begin{cases} TC[RF][iv] & ;\text{option = RF} \\ TC[Spatial][iv] \times TC[RF][iv] & ;\text{option = PE\_Array} \\ TC[Spatial][iv] \times TC[RF][iv] \times TC[SPM][iv] & ;\text{option = SPM} \end{cases}$$

For example, to determine the data allocated in RFs of PEs, we need

$$effective\_TC[iv] = TC[RF][iv] \text{ i.e.,}$$

$$effective\_TC[n] = TC[RF][n] = N\_RF, effective\_TC[fy] = TC[RF][fy] = Fy\_RF, \text{ and so forth.}$$

Then, $data\_alloc[RF][W]$ is calculated by evaluating the indexing expression for operand $W$ where, the value for index $iv$ is used as $effective\_TC[iv]$. Thus, after analyzing index expressions of `W[m][c][fy][fx]`,we get

$$\begin{aligned} data\_alloc[RF][W] &= effective\_TC[m] \times effective\_TC[c] \times effective\_TC[fy] \times effective\_TC[fx] \\ &= M\_RF \times C\_RF \times Fy\_RF \times Fx\_RF \end{aligned}$$

When the RF tiling factors are <1,1,1,1,1,1,3>, registers allocated in a PE for `W` are determined as 1×1 × 1×3 = 3. Similarly, after analyzing indexing expressions of $W$, the model determines the weights communicated to PE array as

$$\begin{aligned} data\_alloc[PE\_Array][W] = {} & [M\_SPATIAL \times M\_RF] \times [C\_SPATIAL \times C\_RF] \\ & \times [Fy\_SPATIAL \times Fy\_RF] \times [Fx\_SPATIAL \times Fx\_RF] \end{aligned}$$

*4.3.2 Estimating Energy Consumption.* Total energy for executing the nested loop consists of the energy consumed in RF accesses, in performing useful operations on PEs, in communicating data via interconnect, and in accessing the data from SPM and DRAM, i.e.,

$$Total\_Energy = e\_Ops + e\_RF + (comm\_energy\_1\_SPM\_pass \times total\_SPM\_pass) + e\_DRAM$$

In our execution model of the dataflow accelerator, during each loop iteration, an operand is read/written from/to RF of a PE for the execution of an operation, i.e.,

$$total\_loop\_iterations = \prod_{i=1}^{total\_IVs} TC[base][IV_i]$$

$$e\_RF = total\_loop\_iterations \times \sum_{op=1}^{total\_Operands} Energy[RF]$$

In the example of Figure 2(b), there are 2 read operands and 1 read+write (r+w) operand. So, the cost for RF accesses during each loop iteration is approximated as 4×$Energy[RF]$. Energy (pJ) of various operations and for accessing data elements from memory are obtained from the literature

Table 3. Notation for Analytical Modeling of Dataflow Execution

| **Term** | **Interpretation** |
|---|---|
| `IV=['n', . . . ,'fx']` | List of loop index variables (from outermost to innermost loop). |
| `total_IVs` | Length of list IV (same as depth of the loop-nest). |
| `level` | Either of {Spatial, RF, SPM, DRAM, base}. |
| `TC[level][iv]` | 2D array of loop iteration counts.<br>For example, TC[RF]['n'] refers to N_RF = 4. |
| `effective_TC[iv]` | Vector of effective loop Trip-Counts per *iv*.<br>Calculated to find the data allocation. |
| `data_alloc[option][op]` | option = {RF, PE_Array, SPM}; op is a data operand. |
| `Data_Reuse[level][op]` | level = {SPM, DRAM}; op is a data operand. |
| `Energy[option]` | option = {RF, Operation_Type, NoC, SPM, DRAM}.<br>Operation_Type corresponds to operations supported by PEs (e.g., MAC, ADD). |

[11, 18] and provided as an input to the model. Moreover, energy for operations performed by PEs is:

$$e\_Ops = total\_loop\_iterations \times \sum_{opr=1}^{total\_Operations} Energy[Operation\_Type[opr]]$$

In the example of Figure 4(b), just 1 Multiply-and-ACcumulate (MAC) operation is performed on a PE in executing a loop iteration. Our model currently does not support loops with conditional statements. However, since each loop iteration sequentially executes on a PE, we plan to extend the model by taking the maximum latency and energy consumption of the true and false paths.

Based on tiling factors for L1 loops, each PE executes a certain number of loop iterations to process the data from allocated registers. We refer it as one *RF pass*. During an RF pass, while PEs process data from RFs, new data for the next RF pass can be accessed from SPM and communicated to PEs via an interconnect network.

$$\begin{aligned} energy\_access\_SPM\_1\_RF\_pass[op] &= data\_alloc[PE\_array][op] \times Energy[SPM] \\ energy\_NOC\_1\_RF\_pass[op] &= data\_alloc[RF][op] \times p[op] \times Energy[NOC] \\ energy\_1\_RF\_pass[op] &= energy\_NOC\_1\_RF\_pass[op] + energy\_access\_SPM\_1\_RF\_pass[op] \end{aligned}$$

Although total data communicated to PE array is determined by `data_alloc[PE_array]`, many PEs may process the same data. We model such spatial reuse by finding the total PEs that read/write the same operand. If an operand *op* belongs to a write operation, we consider only those PEs that produce the outcome. Thus,

$$p[op] = \begin{cases} \prod_{i=1}^{total\_IVs} TC[Spatial][IV_i] & \text{; op belongs to read operation} \\ \prod_{i=1}^{len(list\_dependent\_IV[op])} TC[Spatial][list\_dependent\_IV[op][i]] & \text{; op belongs to write operation} \end{cases}$$

Based on the ordering of the L2 loops (that correspond to SPM accesses), we determine the reuse of data operands for the consecutive RF passes and find communication energy for 1 SPM pass.

$$\begin{aligned} comm\_energy\_1\_SPM\_pass = &\sum_{op=1}^{total\_operands} energy\_1\_RF\_pass[op] \\ &\times (total\_RF\_pass \div Data\_Reuse[SPM][op]) \end{aligned}$$

After determining the data allocated in SPM (processed in 1 SPM pass) and the reuse of the data in SPM, we determine the energy consumption for communicating data from DRAM as follows:

$$e\_DRAM = \sum_{op=1}^{total\_operands} data\_alloc[SPM][op] \times Energy[DRAM] \\ \times (total\_SPM\_pass \div Data\_Reuse[DRAM][op])$$

*4.3.3 Estimating Execution Time.* During processing the data in a RF pass, PEs execute certain number of iterations and perform all operations within each loop iteration. So, estimated cycles are

$$cycles\_UsefulOps = loop\_iterations\_RF\_pass \times latency\_1\_loop\_iteration$$
$$loop\_iterations\_RF\_pass = \prod_{i=1}^{total\_IVs} TC[RF][IVi]$$
$$latency\_1\_loop\_iteration = \sum_{opr=1}^{total\_operations} l$$

$$l = \begin{cases} 1 & \text{; if PE is pipelined} \\ latency[operation_{opr}] & \text{; PE is nonpipelined} \end{cases}$$

During an RF pass, while PEs process data from RFs, new data for the next RF pass can be accessed from SPM and communicated to PEs via interconnect. This interleaving of the communication latency with the computation being performed by PEs can be either achieved by double-buffering the RFs or through software scheduling scheme. If no such support is available, the PE array completely stalls to obtain the necessary data from SPM for the next RF pass. Total cycles required to communicate operands during a RF pass is:

$$comm\_cycles\_operand[op] = data\_alloc[PE\_array][op]/B$$

where B is the width of the data bus for interconnect. Depending on the ordering of L2 loops (that correspond to accessing the data from SPM), some operands are not reused after an RF pass and communicated between SPM and the PE array at every RF pass. However, some operand(s) can be reused and are communicated at every x$^{th}$ RF pass. For example, for an ordering where the loop with index variable c_L2 is innermost, the ofmap $O$ (or the psum) is reused for C_SPM=4 consecutive RF passes. Taking that into account, we determine the communication latency as:

$$comm\_cycles[RF\_pass\#][network\#] = map\_operands\_to\_NOC \\ (comm\_cycles\_operand, Data\_Reuse[SPM])$$

In our default setup, we support popular single-cycle multi-cast interconnect. The networks to communicate read and write operands between SPM and PE array are three and one, respectively [3, 18]. There is one network to communicate r+w operands among PEs (used for reduction operations). Often the total operands in the loop-nest are few and are simultaneously communicated to/from the PEs via interconnect (including executing common kernels like matrix multiplication, convolution, regression, and sequence models). If not, they need to be sequentially broadcast to PEs via available interconnect. For example, when the total data operands are more than available networks, the communication can be scheduled onto networks via a round-robin mechanism. In fact, for performing design space exploration through dMazeRunner, architects can extend the model to accommodate various interconnect topologies. Total cycles required to process the data of SPM (*1 SPM pass*) are:

$$cycles\_SPM\_pass = \sum_{i=1}^{total\_RF\_pass} max(cycles\_UsefulOps, \max_{1 \le j \le total\_networks} comm\_cycles[i][j])$$

Usually, the execution requires several SPM passes. During each SPM pass, the PE array processes the data from one buffer of SPM, and DMA controller accesses DRAM for the data of another buffer. After calculating the size of the data allocated in SPM, we determine total DMA invocations required and the burst size (of contiguous data) per invocation. To calculate DMA cycles, we consider a latency model of Cell processors [44] which featured SPMs, i.e.,

$$DMA\_Model(u) = 291 \text{ (initiation latency)} + 0.24 \times u; \quad u\text{: burst width (bytes)}$$

$$cont\_data\_alloc\_spm[op], DMA\_initiations[op] \leftarrow data\_alloc[SPM][op]$$

$$DMA\_cycles[op] = DMA\_Model(cont\_data\_alloc\_spm[op]) \times DMA\_initiations[op] \\ \times (accel\_freq \div dma\_freq)$$

Based on the data being reused in consecutive SPM passes, we calculate the cycles required for accessing the DRAM during each SPM pass as follows:

$$DRAM\_access\_cycles[SPM\_pass\#] = \sum_{op=1}^{total\_operands} DMA\_cycles[op]$$

$$if(SPM\_pass\# \bmod Data\_Reuse[DRAM][op] == 0)$$

$$total\_cycles = \sum_{i=1}^{total\_SPM\_pass} max(DRAM\_access\_cycles[i], cycles\_SPM\_pass)$$

Note: Implementation of our execution model deals with the various complex scenarios including:

- *Modeling stall cycles and energy consumption for inter-PE communication:* When a r+w operand (e.g., *O*) is invariant of a loop (*c*, `fy`, `fx`) that executes spatially (e.g., `C_SPATIAL` > 1), computing the output requires inter-PE communication. Depending on the data buffering mechanism of the RF, PE array may not start processing new data from RF or cannot get new data from interconnect while PEs perform the reduction operations onto previously computed data. Therefore, depending on the spatial execution of loops and data reuse factors, stall cycles and energy consumed are accounted.
- *Accurate modeling of continuous data reuse through several RF+SPM passes:* Depending on the ordering of the loops, some operand gets reused continuously throughout all RF passes of an SPM pass and through several such SPM passes. For example, for an ordering of L2 loops with IVs {`n_L2,m_L2,oy_L2,ox_l2,c_L2,fy_L2,fx_L2`} (outermost to innermost) with TCs <1,1,1,1,4,3,3>, total RF passes in an SPM pass are 4×3×3 = 36. In each RF pass, operands *I* and *W* are communicated from SPM to RFs via NoC while *O* is reused in RFs. Now, for an ordering of L3 loops with IVs {`oy_L3,ox_l3,fy_L3,fx_L3, n_L3,m_L3,c_L3`} with TCs <1,1,1,1,2,32,16>, *O* gets reused in consecutive 16 SPM passes. Thus, write-back of *O* occurs just once after every 16 SPM passes; each SPM pass consists of 36 RF passes. We refer to such reuse of data at consecutive memory levels as a continuous reuse and accurately model it for various operands.
- *Detailed model of data reuse and communication for r+w operands:* Processing of a r+w operand on PE array may require to read previously computed value (e.g., input psum) from SPM and interconnect. Furthermore, such read operation can be skipped some times,

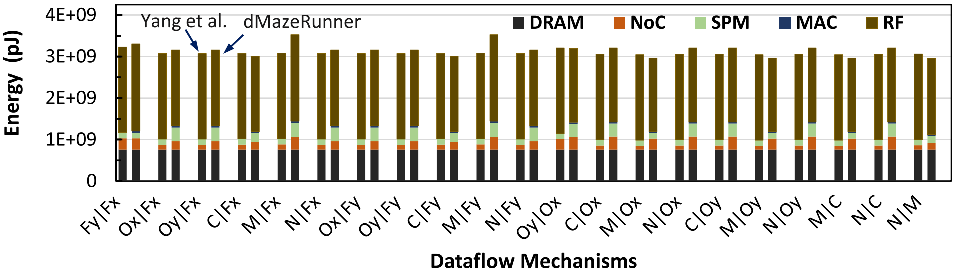


Fig. 7. Validation results for ResNet conv5_2. The energy consumption estimated by dMazeRunner is close to the energy model of [18, 24].

if the operand is zero-initialized. Thus, we offset the calculation of the miss penalty and energy consumption accordingly.

## 5 VALIDATION EXPERIMENTS

**Specification of the target platform:** We considered a similar dataflow accelerator architecture as recent works [3, 5, 18]. The accelerator consists of 16×16 PEs with 16-bit precision. Each PE accesses 512B RF and a 128 kB scratch-pad. Like Eyeriss architecture [11], each pipelined PE consists of a 2-stage multiplier and an adder. The accelerator features 4 single-cycle multi-cast networks [3] to communicate the operands to PEs and 1 such network for reduction. The SPM consists of 64 banks (2 kB each) that can be allocated to any data. Data is accessed from DRAM via DMA and managed in SPM with double-buffering [41, 45]. Our latency model for data transfers via DMA is same as Cell processors that featured SPMs [44]. Energy costs for accelerator resources were obtained from hardware evaluations by Yang et al. [18] for a 28 nm technology.

**Validation against Yang et al. [24]:** To determine the accuracy of our dataflow execution model, we validate it against evaluations of a recent work [18, 24]. Validating the execution model is often challenging since it requires (i) the same architecture specification, (ii) the information about the adopted execution method, and (iii) the absolute values of execution time and energy consumed by the platforms. Therefore, we used the execution methods obtained by the tool [24] and evaluated the same methods through the analytical model of dMazeRunner.

This validation experiment covers various *dataflow mechanisms* which represent *how different loops are executed spatially*. For example, `Fy|Fx` represents a weight stationary mechanism where PEs are grouped based on unrolling Fy and Fx loops for spatial execution [18]. Note that these dataflow mechanisms also incorporate the variations in temporal execution (different data reuse patterns) and the spatial execution of more than two loops.

We find that dMazeRunner achieves the same PE utilization as Yang et al. [18, 24]. Moreover, Figure 7 shows that for various dataflow mechanisms, the energy consumption (in pJ) estimated by dMazeRunner closely matches to the energy estimation tool [24] (the difference is 4.19%). In fact, for commonly used dataflow mechanisms like output-stationary (`Oy|Ox`), the difference is 0.3%. We observe a higher difference (about 14%) for $M|Fx$ mechanism. A possible reason is that the model of [24] is more accurate for the interconnect organization (e.g., per-hop communication cost) while dMazeRunner considers the same cost for multicast communication.

Furthermore, Figure 7 shows the breakdown of the energy for system resources where, each bar on the left-hand side represents the evaluation from Yang et al. [24], and the second bar for each mechanism represents the estimation from dMazeRunner. We find that energy estimated for system resources is similar to that obtained by [24]. In fact, for optimized execution methods, the

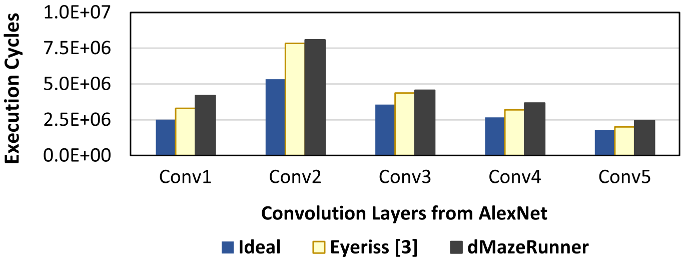


Fig. 8. Performance validation against Eyeriss architecture.

estimated energy for DRAM accesses is very low, and most of the energy consumption is attributed to accessing data from RFs.

**Validation against Eyeriss architecture:** We extended our dataflow execution model for modeling Eyeriss accelerator [3] which executed AlexNet for ImageNet classification [19]. We evaluated execution methods reported in [3]. We considered following Eyeriss-specific enhancements for the model: (i) separate and larger bit-widths of different input, output, and reduction networks, (ii) a dedicated mesh-style network for reduction, and (iii) in communicating the data (e.g., a row of 1×3 ifmap), reusing the neighborhood data (1×2 ifmap) from RFs in the sliding-window execution. Figure 8 shows the execution cycles considering (a) ideal acceleration (i.e., total MAC operations ÷ total PEs), (b) processing time reported for Eyeriss architecture [3], and (c) estimation of execution cycles. We find that our estimations closely matched execution cycles of the architecture [3], with a difference of 11% in the total execution cycles.

## 6 EXPERIMENT RESULTS AND ANALYSIS

**Benchmarks:** For evaluating different execution methods (featuring diverse data reuse patterns and various ways of spatial execution), we consider different convolution layers from widely used DNNs ResNet and ResNext [1, 20] for ImageNet classification (with batch size of 4 images). We use the same target architecture as energy validation experiments.

**Techniques evaluated:** To evaluate the effectiveness of the optimal solutions achieved by dMaze-Runner, we determine various execution methods for dataflow mechanisms described by previous techniques: (i) For output stationary mechanism (`Oy|Ox`), (i.a) **SOC** [11, 12] in which, entire PE array processes *single output channel*, and (i.b) simultaneous processing of *multiple output channels* **(MOC)** [11, 38] on different PE-groups for *ifmap reuse*. For both SOC and MOC, the data movement schedule iterates over channels for minimizing *psum* accumulation cost, (ii) **WS1** for weight stationary mechanism (`Fy|Fx`) [5, 11], (iii) **RS** [3] for row stationary (`Oy|Fy`) mechanism, which maximizes weight reuse in RF, psum accumulation in RFs/PE-array, psum reuse in SPM, and (iv) coarse weight stationary **(WS2)** for `M|C` mechanism, which is like matrix-multiplication on systolic arrays [2]. We are not aware of any previous technique that optimizes EDP through other dataflow mechanisms. However, we evaluate all mechanisms to demonstrate effectiveness of achieved execution methods.

### 6.1 dMazeRunner Outperforms Prior Execution Schemes and Reduces EDP by 9.16×

Figure 9(a) shows the evaluation of various execution methods for popular dataflow mechanisms. The evaluations depict EDP of each convolution layer on the primary axis and total execution cycles for these six layers on the secondary axis (lower the better). For better visualization, we plot EDP results on a logarithmic scale.

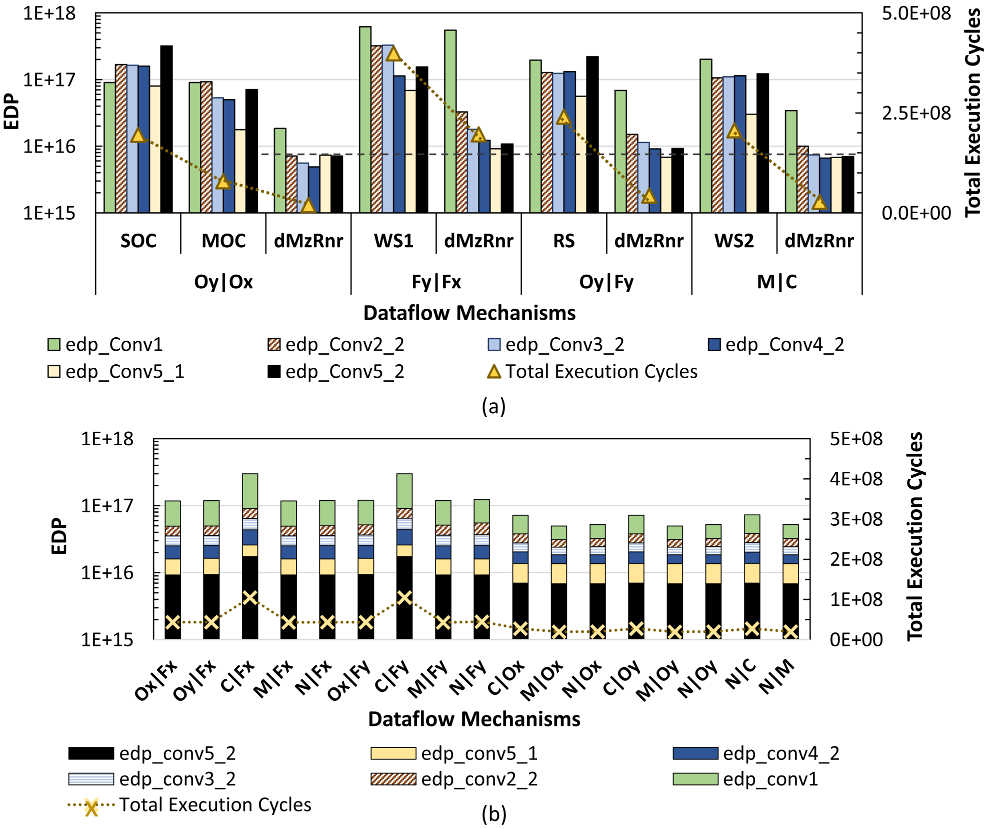


Fig. 9. (a) For popular mechanisms, dMazeRunner achieved execution methods reduce the total EDP by 9.16× on average. (b) dMazeRunner also achieves the optimal execution methods for other dataflow mechanisms.

**Observation (i)** *For each dataflow mechanism, dMazeRunner significantly reduced the EDP and total cycles, when compared to execution methods achieved by prior approaches.* For example, for conv5_2, dMazeRunner reduced EDP by 44.47× and execution cycles by 18.72×, as compared to SOC. On average, dMazeRunner reduced the total EDP of convolution layers by 9.16× over other techniques and execution cycles by 5.83×. The primary reason for such a significant scope of improvement is that *prior techniques target certain ways of spatial execution and data reuse, which are often, not very efficient.* For example, when 14×14 PEs executed ofmaps or the output channel(s) spatially, the PE utilization achieved on a 16×16 array was just 76%. Similarly, SOC and MOC maximized psum accumulation in RF, which did not always yield high RF utilization (e.g., 436B utilized for 512B RF). Moreover, *with a fixed optimization strategy to reuse certain data operand(s), no single heuristic efficiently leveraged the maximum data reuse possible.* For example, for convolution layers at beginning of ResNe(x)t (*conv1*), ifmaps are significantly larger and *weight reuse* is desired. In contrast, for later layers (*conv4_2*), weights dominate the data movement, and *ifmap reuse* yields better execution. *The execution methods obtained by prior approaches were not able to adapt to such dynamics of loop characteristics.* Thus, prior techniques neither ensured very high resource utilization, nor efficient reuse of all data operands. So, even if they somehow obtained a reasonable solution, a scope for further reduction in both execution time and energy remained. With holistic representation, dMazeRunner captured the vast space of execution methods and after drastically pruning the search, dMazeRunner made the brute-force exploration feasible. Consequently, it achieved the optimal execution methods that outperformed prior techniques.

**Observation (ii)** *dMazeRunner generated execution methods achieved various data reuses at different accelerator resources and minimized DRAM accesses for various operands:* With a certain optimization strategy, prior heuristics leveraged reuse of specific operands. For example, *SOC* and *MOC* maximized psum reuse at RF and SPM levels, while *RS* maximized weight reuse in RFs and psum reuse in SPM. For executing conv5_1 layer with `Oy|Ox` mechanism, *SOC* allocated 28kB ifmaps, 18kB weights, and 1.5kB psum in SPM, which were accessed from DRAM 512, 512, and 128 times, respectively. This resulted in DRAM access of 14.74MB, 9.44MB, and 0.2MB, respectively. However, the execution method of dMazeRunner allocated 14.06kB of ifmaps, 18kB weights, and 12.25kB psum in SPM, which were accessed from DRAM 256, 256, and 16 times. So, dMazeRunner accessed DRAM for 3.68MB ifmaps, 4.7MB weights, and 0.2MB psum. Thus, dMazeRunner obtained solution exhibited a better choice of tiling factors and minimized total DRAM accesses for ifmaps by 4× and 2× for weights. In fact, it maximized *ifmap* and *filter reuse* spatially, *convolution reuse* in RF, and *psum reuse* at RF and SPM levels. Similarly, for executing conv5_2 layer with `M|C` mechanism, it reduced DRAM accesses by 16× for ofmap, as compared to WS2. By significantly reusing all operands, execution methods of dMazeRunner minimized DRAM accesses, reducing both the energy and execution cycles. Thus, although the acceleration gains during chip execution can differ from estimations, through better data reuse, reduced DRAM accesses, and efficient interleaving of computation with communication, dMazeRunner achieved solutions can outperform prior heuristics.

**Observation (iii)** *With holistic exploration, dMazeRunner achieved the optimal solutions which yield similar EDP and execution time for various dataflow mechanisms:* Figure 9(a) shows that for various mechanisms, the achieved solutions result in a very similar EDP and execution time (note the dotted line). This is because: (i) for efficient acceleration, often more than two loops are spatially executed (e.g., *M* and *C* along with *Oy* and *Ox*) and hence, two mechanisms may attain the same solution, and (ii) highly efficient solutions share common characteristics like high utilization of resources, maximized reuse of various operands, efficient interleaving of computation with communication (i.e., minimum to no miss penalty). Therefore, for individual mechanisms, the achieved optimal solutions yield similar results. Moreover, Figure 9(b) depicts the EDP and execution cycles for 17 more mechanisms and demonstrates similar results. However, when reduction operations are performed through inter-PE communication, it results in higher cycles in our model. This is because, we targeted one single-cycle multi-cast network for r+w operands instead of a mesh-style interconnect. This is reflected in a relatively high execution time and EDP for mechanisms like `Fy|Fx`, `C|Fx`, and `Ox|Fy`. We can observe such difference at least for conv1 layer, which consisted of larger feature maps. Note that none of the prior heuristics pruned the space such drastically that a brute-force algorithm is applied to achieve the optimal solutions. Furthermore, no prior technique achieved the optimal solutions that minimize EDP while using a variety of dataflow mechanisms. Therefore, Figure 9(b) does not feature any evaluations of prior works.

## 6.2 dMazeRunner Reduces Energy Consumed for Dataflow Execution up to 30.84%

Recently [18] proposed an auto-optimizer [24] to reduce the energy consumption of DNN dataflow execution. We executed the various convolution layers of ResNet with [24] and obtained optimized execution methods. We evaluate them with the solutions achieved by dMazeRunner and demonstrate the impact of holistic exploration.

**Observation (iv)** *Drastic pruning enabled exhaustive exploration for achieving the optimal execution methods:* Figure 10 shows the energy consumption of optimized methods obtained by [24] and that of dMazeRunner. Here, the energy of a convolution layer is obtained from the best outcome

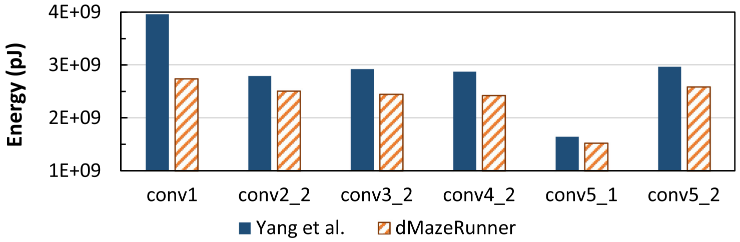


Fig. 10. dMazeRunner reduces energy consumption up to 30.84% as compared to auto-optimizer of [24].

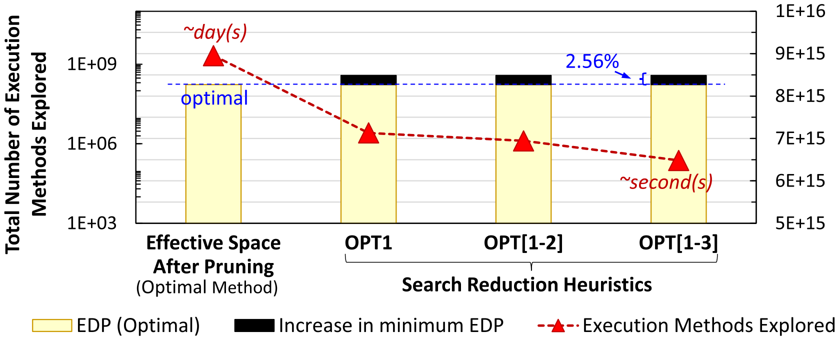


Fig. 11. dMazeRunner reduces execution methods explored by 9020× and achieves highly efficient execution methods (2.56% increase in the minimum EDP) in seconds.

among all execution methods explored. Execution methods achieved by dMazeRunner outperformed [24] by reducing the energy up to 30.84% and by 15.55% on average. Even for individual dataflow mechanisms (like output- or row-stationary [3, 18]), dMazeRunner reduced the energy significantly over [24]. For example, dMazeRunner reduced the energy of executing conv4_2 with N|Ox by 36% and conv5_2 with Oy|Ox by 14%. For previous techniques like [18, 24, 37], it is infeasible to explore the space exhaustively. For example, for a configuration of tiling factors, when optimizer of [24] determines loop orderings, it considers 4! combinations when iteration counts of four loops are greater than unity. Therefore, to make the exploration feasible, [18, 24] heuristically considered a tiny fraction of the space. For example, for conv5_2 and conv4_2, [24] explored 1976 and 4608 methods (with different tiling factors) and multiple orderings per method. Similarly, for conv2_2, [24] explored 6448 methods in 8.8 hours. On the other hand, since dMazeRunner drastically pruned the orderings, it required to evaluate up to 3×3 orderings per method (Table 2 + OPT4), as compared to up to 7!×7! (25 million) orderings. For example, for conv5_2 and conv4_2, dMazeRunner exhaustively explored 1.75E+07 and 1.8E+08 execution methods (with varying tiling factors) and determined the optimal execution methods.

Note that heuristically exploring a small fraction of all execution methods may not yield efficient EDP or reasonable execution time. In fact, heuristically obtained solution may fail to efficiently interleave the computation with communication latency (high miss penalty). Since [18, 24] lacked performance model, we are unable to compare the EDP or execution time of our methods with it.

### 6.3 dMazeRunner Achieves Close-to-Optimal Solutions in Seconds

Figure 11 shows the total execution methods evaluated for the convolution layers and the total EDP. For achieving the optimal solutions, dMazeRunner pruned the space and exhaustively evaluated a total of 2.12E+09 methods. For example, when executed on an Intel i7-6700 quad-core

Table 4. Layer (Kernel) Specific and Overall Best Memory Sizes for a 256-PE Accelerator

| ResNet Layer | Layer-Specific Best Design | EDP | Overall Top #1 Design | EDP (normalized) | Overall Top #2 Design | EDP (normalized) |
|---|---|---|---|---|---|---|
| Conv1 | <256,512k> | 1.59E+16 | <256, 512k> | 1x | <256, 1024k> | 1.16x |
| Conv2_2 | <256,512k> | 4.52E+15 | | 1x | | 1.03x |
| Conv3_2 | <256,1024k> | 3.56E+15 | | 1.10x | | 1x |
| Conv4_2 | <256,1024k> | 3.57E+15 | | 1.11x | | 1x |
| Conv5_1 | <256,1024k> | 2.83E+15 | | 1.08x | | 1x |
| Conv5_2 | <256,128k> | 6.14E+15 | | 1.05x | | 1.03x |
| Total (6 layers) | <256,512k> | 3.78E+16 | | 1x | | 1.04x |

platform, dMazeRunner evaluated 1.75E+07 methods for conv5_2 in several tens of minutes and required several hours to achieve the optimal execution method for conv4_2.

**Observation (v)** *Pruning heuristic can achieve the solutions in second(s) with a negligible increase in the minimum EDP:* After incrementally applying our pruning heuristics (OPT1, OPT2 and OPT3, which were described in Section 4.2.3), we observe that dMazeRunner drastically reduced total methods evaluated and consistently achieved close-to-optimal solutions. For example, Figure 11 shows that OPT1 reduced total methods to 2.57E+06, at the cost of a mere 2% increase in the total EDP of the optimal solution. For OPT1, we set utilization factors as 80% for PEs, 80% for RF, and 50% for SPM, which ensured that the majority of solutions discarded are inefficient ones. Then, OPT2 discarded the solutions that resulted in non-contiguous data accesses (potential candidates for incurring high miss penalty). Finally, OPT3 discarded the execution methods that required inter-PE communication, from a total of 1.28E+06 (for OPT1+OPT2) to 2.35E+05. Thus, with OPT[1–3], dMazeRunner reduced the search by 9020× at the cost of a 2.56% increase in the minimum EDP. The same is true for individual layers. For example, for conv5_2, dMazeRunner reduced the total methods from 1.75E+07 to 753 (with the same EDP as the optimal solution) and from 1.42E+07 to 877 for conv5_1 (with a 5% increase in the minimum EDP). Thus, while exhaustively exploring the optimal solution for the application can require processing over several days (or hours for individual layers), dMazeRunner achieved close-to-optimal solutions in just a few seconds (1 second for conv5_2 and a maximum of 122 seconds for optimizing conv2_2).

Note that OPT4 (considering only those methods that maximize data reuse) and OPT5 (leveraging multi-threading and caching) are also applicable to an exhaustive search. So, we enabled them for all evaluations of dMazeRunner. Since there is no one-size-fits-all solution, heuristics employing a specific execution method or randomly exploring the tiny space do not always yield very efficient solutions. However, well-crafted pruning heuristics promptly obtain a set of methods that exhibit EDP close to the minimum.

## 6.4 Design Space Exploration

dMazeRunner can be leveraged to invoke a rapid DSE for landing upon better architectural design solutions. Table 4 lists the results of a DSE experiment which optimizes the on-chip memory sizes for the targeted 256-PE accelerator. The second-left column lists the best memory configuration for each layer and the third-left column lists corresponding EDP. The columns on the right-hand side show the two best designs that achieved the best EDP (normalized) for some layers and in total, a lower EDP as compared to other configurations. Both the designs #1 and #2 (in fact, the top four designs) featured 256B RFs per PE. Figure 12 depicts the EDP (a total of all the six convolution kernels) for the variations in the RF sizes (primary horizontal axis) and SPM sizes (series in the legend).

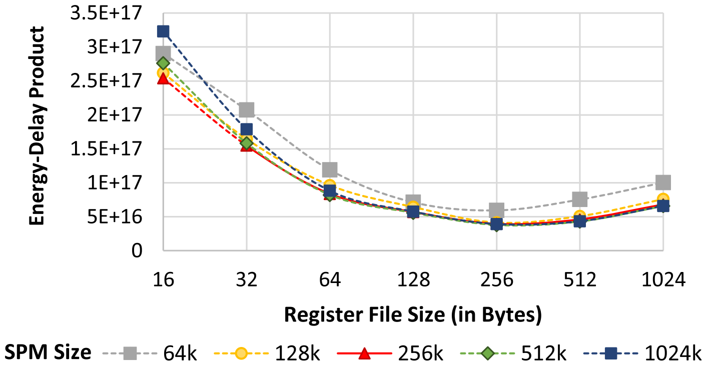


Fig. 12. DSE of memory sizes for a 256-PE accelerator. Designs featuring 256B RFs and 512kB SPMs yield lower EDP.

Figure 12 shows that EDP is notably lower when the RF size is 128B or larger. This is because, the convolution kernels exhibit the significant reuse of different operands, which can be better sustained with larger RFs, avoiding costly accesses to SPM and DRAM. However, increasing the RF size beyond 512B increases the EDP again, since it is hard to efficiently utilize RF (i.e., finding a schedule that balances communication latency with computation from the RF) while the energy cost to access RF increases. The RF size of 256B demonstrates a balance in the trade-off and yield significantly lower EDPs. Similarly, an SPM size of 512kB demonstrates lower EPDs. If an SPM size is relatively small (e.g., 64kB or smaller) then, after reusing the data at the RF level, the accesses mostly go to DRAM since there is little-to-no reuse at SPM level. On the other hand, accessing a larger SPM significantly costs more energy and can yield a large increase in the EDP, if RF size is smaller (e.g., consider a 512kB or 1024kB SPM and a 16B RF). Thus, dMazeRunner can be leveraged for exploring the optimized designs.

## 7 FUTURE WORK

Future work targets exploring the efficient designs and optimized mappings for various important machine learning, imaging and media kernels, including employing application specific optimizations like exploiting sparsity, model-level optimizations, and inter-layer data reuse in DNNs.

## 8 CONCLUSIONS

For efficient spatiotemporal execution of perfectly nested loops on dataflow accelerators, it is crucial to determine highly efficient execution method that minimizes EDP by achieving high utilization of resources, maximized reuse of various operands, and efficient interleaving of computation with communication latency. Due to the vast space of execution methods, there is a lack of a comprehensive solution that can accurately explore all the execution methods and efficiently map loops on dataflow accelerators. We proposed dMazeRunner, which formulates a *holistic representation to inform the optimizer about the vast space of various execution methods*. Then, dMazeRunner drastically prunes the space by constructing valid methods that feature unique data reuses and *exhaustively explores the optimal solution*. Furthermore, dMazeRunner employs pruning heuristics to achieve close-to-optimal solutions *in a few seconds*. Finally, *the analytical model of dMazeRunner enables static estimation of the efficacy of an execution method*, which helps the exploration of loop optimizations and the design space. Compared to prior approaches, dMazeRunner achieved solutions reduced the total EDP by 9.16× and the total execution cycles by 5.83×. Moreover, search-space reduction heuristics of dMazeRunner reduced the exploration of execution methods by over

9000× with a mere 2.56% increase in EDP, as compared to the optimal mapping. With a systematic, succinct, and automated exploration, dMazeRunner alleviates the burden of the programmers and architects to manually fine-tune the mapping among the vast space and can be leveraged to explore the optimized designs of dataflow accelerators.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback and suggestions and Mr. Sagar Parekh at Compiler Mircoarchitecture Lab, ASU for assisting in automation of some evaluations. This research was partially supported by funding from National Science Foundation under grant CCF 1723476 - NSF/Intel joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA), and from the grants NRF-2015M3C4A7065522 (Next-generation Information Computing Development Program, funded by National Research Foundation of Korea, MSIT) and 2014-3-00035 (Research on High Performance and Scalable Manycore Operating System, funded by IITP, MSIT). Any opinions, findings, and conclusions presented in this material are those of the authors and do not necessarily reflect the views of their employers or the sponsoring agencies.

## REFERENCES

[1] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1492–1500.

[2] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–12.

[3] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2016), 127–138.

[4] HT Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 821–834.

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

[6] Hongbo Rong. 2017. Programmatic control of a compiler for generating high-performance spatial hardware. *arXiv preprint* arXiv:1711.07606 (2017).

[7] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2018. Scale-sim: Systolic cnn accelerator. *arXiv preprint* arXiv:1811.02883 (2018).

[8] Michael Pellauer, Angshuman Parashar, Michael Adler, Bushra Ahsan, Randy Allmon, Neal Crago, Kermin Fleming, Mohit Gambhir, Aamer Jaleel, Tushar Krishna, et al. 2015. Efficient control and communication paradigms for coarse-grained spatial architectures. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 10.

[9] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2013. A general constraint-centric scheduling framework for spatial architectures. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 495–506.

[10] Yang You, Zhao Zhang, Cho-Jui Hsieh, Jim Demmel, and Kurt Keutzer. 2019. Fast deep neural network training on distributed systems and cloud TPUs. *IEEE Transactions on Parallel and Distributed Systems* (2019).

[11] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 367–379.

[12] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 92–104.

[13] Shouyi Yin, Peng Ouyang, Shibin Tang, Fengbin Tu, Xiudong Li, Shixuan Zheng, Tianyi Lu, Jiangyuan Gu, Leibo Liu, and Shaojun Wei. 2017. A high energy efficient reconfigurable hybrid neural network processor for deep learning applications. *IEEE Journal of Solid-State Circuits* 53, 4 (2017), 968–982.

[14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-end optimization stack for deep learning. *arXiv preprint* arXiv:1802.04799 (2018), 1–15.

[15] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 16–25.

[16] SCALE-Sim. https://github.com/ARM-software/SCALE-Sim. ([n. d.]). Accessed: November 5, 2018.

[17] Ye Yu, Yingmin Li, Shuai Che, Niraj K Jha, and Weifeng Zhang. 2019. Software-defined design space exploration for an efficient AI accelerator architecture. *arXiv preprint* arXiv:1903.07676 (2019).

[18] Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qiaoyi Liu, Steven Emberton Bell, Jeff Ou Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, et al. 2018. DNN dataflow choice is overrated. *arXiv preprint* arXiv:1809.04070 (2018).

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 1097–1105.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[21] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. 2018. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint* arXiv:1811.09886 (2018).

[22] Yann LeCun. 2019. 1.1 deep learning hardware: Past, Present, and Future. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 12–19.

[23] Kartik Hegde, Rohit Agrawal, Yulun Yao, and Christopher W. Fletcher. 2018. Morph: Flexible acceleration for 3D CNN-based video understanding. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 933–946.

[24] Xuan Yang et al. DNN Energy Model and Optimizer. https://github.com/xuanyoya/CNN-blocking/tree/dev. Accessed: November 5, 2018.

[25] Bruce Fleischer, Sunil Shukla, Matthew Ziegler, Joel Silberman, Jinwook Oh, Vijavalakshmi Srinivasan, Jungwook Choi, Silvia Mueller, Ankur Agrawal, Tina Babinsky, et al. 2018. A scalable multi-TeraOPS deep learning processor core for AI trainina and inference. In *2018 IEEE Symposium on VLSI Circuits*. IEEE, 35–36.

[26] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[27] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2019. HyPar: Towards hybrid parallelism for deep learning accelerator array. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 56–68.

[28] Alfred V. Aho et al. 2007. Compilers: Principles, techniques and tools. (2007).

[29] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. *Compiler Optimizations for Improving Data Locality*. Vol. 29. ACM.

[30] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. 2007. DRDU: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 12, 2 (2007), 15.

[31] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Acm Sigplan Notices*, Vol. 43. ACM, 101–113.

[32] Florin Balasa, Per Gunnar Kjeldsberg, Arnout Vandecappelle, Martin Palkovic, Qubo Hu, Hongwei Zhu, and Francky Catthoor. 2008. Storage estimation and design space exploration methodologies for the memory management of signal processing applications. *Journal of Signal Processing Systems* 53, 1–2 (2008), 51.

[33] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[34] Bernhard Egger, Hochan Lee, Duseok Kang, Mansureh S. Moghaddam, Youngchul Cho, Yeonbok Lee, Sukjin Kim, Soonhoi Ha, and Kiyoung Choi. 2017. A space-and energy-efficient code compression/decompression technique for coarse-grained reconfigurable architectures. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 197–209.

[35] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. 2018. Ramp: Resource-aware mapping for cgras. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[36] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. 2018. URECA: A compiler solution to manage unified register file for CGRAs. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1081–1086.

[37] Arthur Stoutchinin, Francesco Conti, and Luca Benini. 2019. Optimally scheduling CNN convolutions for efficient memory access. *arXiv preprint* arXiv:1902.01492 (2019).

[38] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *International Conference on Machine Learning*. 1737–1746.

[39] Zhongyuan Zhao, Hyoukjun Kwon, Sachit Kuhar, Weiguang Sheng, Zhigang Mao, and Tushar Krishna. 2019. mRNA: Enabling efficient mapping space exploration for a reconfiguration neural accelerator. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 282–292.

[40] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 161–170.

[41] Hyoukjun Kwon, Michael Pellauer, and Tushar Krishna. 2018. MAESTRO: An open-source infrastructure for modeling dataflows within deep learning accelerators. *CoRR* abs/1805.02566 (2018).

[42] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. IEEE Computer Society, 75.

[43] fmincon. https://www.mathworks.com/help/optim/ug/fmincon.html. Accessed: November 5, 2018.

[44] Michael Kistler, Michael Perrone, and Fabrizio Petrini. 2006. Cell multiprocessor communication network: Built for speed. *IEEE Micro* 26, 3 (2006), 10–23.

[45] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, Jonghee W. Yoon, Doosan Cho, and Yunheung Paek. 2011. High throughput data mapping for coarse-grained reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 11 (2011), 1599–1609.

Received April 2019; revised June 2019; accepted July 2019