2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00050

CoSA: Scheduling by Constrained Optimization for Spatial Accelerators

Qijing Huang
UC Berkeley

Minwoo Kang
UC Berkeley

Grace Dinh
UC Berkeley

Thomas Norell
UC Berkeley

Aravind Kalaiah
Facebook

James Demmel
UC Berkeley

John Wawrzynek
UC Berkeley

Yakun Sophia Shao
UC Berkeley

***Abstract*—Recent advances in Deep Neural Networks (DNNs) have led to active development of specialized DNN accelerators, many of which feature a large number of processing elements laid out spatially, together with a multi-level memory hierarchy and flexible interconnect. While DNN accelerators can take advantage of data reuse and achieve high peak throughput, they also expose a large number of runtime parameters to the programmers who need to explicitly manage how computation is scheduled both *spatially* and *temporally*. In fact, different scheduling choices can lead to wide variations in performance and efficiency, motivating the need for a fast and efficient search strategy to navigate the vast scheduling space.**

To address this challenge, we present CoSA, a constrained-optimization-based approach for scheduling DNN accelerators. As opposed to existing approaches that either rely on designers' heuristics or iterative methods to navigate the search space, CoSA expresses scheduling decisions as a constrained-optimization problem that can be deterministically solved using mathematical optimization techniques. Specifically, CoSA leverages the regularities in DNN operators and hardware to formulate the DNN scheduling space into a mixed-integer programming (MIP) problem with algorithmic and architectural constraints, which can be solved to automatically generate a highly efficient schedule in one shot. We demonstrate that CoSA-generated schedules significantly outperform state-of-the-art approaches by a geometric mean of up to 2.5× across a wide range of DNN networks while improving the time-to-solution by 90×.

***Index Terms*—scheduling, accelerator, neural networks, compiler optimizations**

I. Introduction

Deep neural networks (DNNs) have gained major interest in recent years due to their robust ability to learn based on large amounts of data. DNN-based approaches have been applied to computer vision [34], [43], [57], machine translation [64], [68], audio synthesis [66], recommendation models [31], [46], autonomous driving [11] and many other fields. Motivated by the high computational requirements of DNNs, there have been exciting developments in both research and commercial spaces in building specialized DNN accelerators for both edge [1], [16], [17], [26], [50], [61], [63], [72] and cloud applications [5], [19], [27], [36], [39], [69].

State-of-the-art DNN accelerators typically incorporate large arrays of processing elements to boost parallelism, together with a deep multi-level memory hierarchy and a flexible network-on-chip (NoC) to improve data reuse. While these architectural structures can improve the performance and energy efficiency of DNN execution, they also expose a large number of scheduling parameters to programmers who must decide when and where each piece of computation and data movement is mapped onto the accelerators both spatially and temporally. Here, we use *schedule* to describe how a DNN layer is partitioned spatially and temporally to execute on specialized accelerators. Given a target DNN layer and a specific hardware architecture, there could be millions, or even billions, of valid schedules with a wide range of performance and energy efficiency [49]. Considering the vast range of DNN layer dimensions and hardware architectures, there is a significant demand for a generalized framework to quickly produce efficient scheduling options for accelerators of varying hardware configurations.

Achieving high performance on a spatially distributed architecture requires several factors to be carefully considered, including tiling for good hardware utilization, pipelining data movement with compute, and maximizing data re-use. Previous scheduling frameworks have attempted to reflect these considerations by formulating an analytical cost model, pruning the scheduling space with known hardware constraints, and then exhaustively searching for the best candidate based on their cost models [14], [23], [49], [71]. However, navigating the scheduling space in such a brute-force fashion can easily become intractable for larger DNN layers and more complex hardware architectures. Other notable efforts have employed feedback-driven approaches, such as black-box tuning, beam search, and other machine learning algorithms with iterative sampling [3], [15], [38]. However, these schedulers typically require massive training datasets and large-scale simulations to learn performance models, making it infeasible to extend them to other types of hardware accelerators, especially those still under development. Hence, there is a clear need for efficient scheduling mechanisms to *quickly* navigate the search space and produce *performant* scheduling options.

In this work, we demonstrate CoSA, a constrained-optimization-based approach to schedule DNN accelerators. In contrast to prior work that either requires exhaustive brute-force-based or expensive feedback-driven approaches, CoSA expresses the DNN accelerator scheduling as a constrained-optimization problem that can be deterministically solved using

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00050

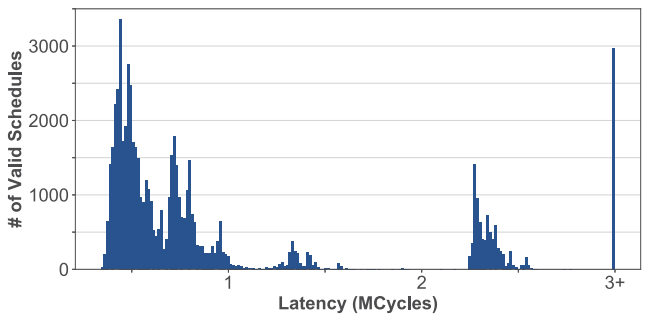


Fig. 1: Execution latency histogram of 40K valid scheduling choices for a ResNet-50 layer on a spatial accelerator.

today's mathematical optimization libraries in one pass. In particular, CoSA leverages the regularities in both DNN layers and spatial hardware accelerators where the algorithmic and hardware parameters can be clearly defined as scheduling constraints. Specifically, CoSA formulates the DNN scheduling problem as a prime-factor allocation problem that determines 1) tiling sizes for different memory levels, 2) relative loop ordering to exploit reuse, and 3) how computation should be executed spatially and temporally. CoSA constructs the scheduling constraints by exposing both the algorithmic behaviors, e.g., layer dimensions, and hardware parameters, e.g., memory and network hierarchies. Together with clearly defined and composable objective functions, CoSA can solve the DNN scheduling problem in one shot without expensive iterative search. Our evaluation demonstrates that CoSA-generated schedules outperform state-of-the-art approaches by $2.5\times$ across different DNN network layers, while requiring $90\times$ less scheduling time as it does not require iterative search.

In summary, this work makes the following contributions:

- We formulate DNN accelerator scheduling as a constrained-optimization problem that can be solved in a single pass. To the best of our knowledge, CoSA is the first constrained-optimization-based approach to tackle major DNN scheduling decisions in one shot.
- We take a communication-oriented approach in the CoSA formulation that highlights the importance of data transfer across different on-chip memories and exposes the cost through clearly defined objective functions.
- We demonstrate that CoSA can quickly generate high-performance schedules outperforming state-of-the-art approaches for different DNN layers across different hardware architectures.

II. Background and Motivation

In this section, we discuss the complexity of DNN scheduling space and the state-of-the-art schedulers to navigate the space.

A. DNN Scheduling Space

Scheduling is a crucial decision-making process for the compilers to effectively assign workload to compute resources. With the emergence of numerous DNN accelerators with diverse architectures, there is a need for a fast, performant, and explainable approach to scheduling. Our work focuses on operator-level scheduling, which aims to optimize the performance of each operator, i.e. DNN layer, on specific hardware. Operator-level scheduling typically comprises three key loop optimizations: *loop tiling*, *loop permutation*, and *spatial mapping*. *Loop tiling* describes which loops are mapped to which memory hierarchy and the corresponding tile sizes. *Loop permutation* determines the relative order of the loops, while *spatial mapping* binds one or more loop dimensions to spatial hardware resources, such as parallel processing elements, instead of mapping them to temporal (i.e. sequential) execution. Each optimization can have a significant impact on the performance, and all three optimizations need to be considered together to achieve the best performance.

Scheduler	Search Algorithm
Brute-force Approaches:	
Timeloop [49]	Brute-force & Random
dMazeRunner [23]	Brute-force
Triton [65]	Brute-force over powers of two
Interstellar [71]	Brute-force
Marvel [14]	Decoupled Brute-force
Feedback-based Approaches:	
AutoTVM [15]	ML-based Iteration
Halide [56]	Beamsearch [3], OpenTuner [6], [45]
FlexFlow [38]	MCMC
Gamma [40]	Genetic Algorithm
Mind Mapping [35]	Gradient-based Search
Constrained Optimization Approaches:	
Polly+Pluto [12], [13], [30]	
Tensor Comprehension [67]	Polyhedral Transformations
Tiramisu [8]	
CoSA	**Mixed-Integer Programming (MIP)**

TABLE I: State-of-the-art DNN accelerator schedulers.

Consider scheduling a 3×3 convolution layer in ResNet50 [34] with 256 input and output channels, and an output dimension of 14×14, on an accelerator with five levels of memory. If we split each individual loop bound into its prime factors and assign each one to a memory level, we would have billions of schedules to consider. Among the randomly sampled schedules from all possible loop tilings, half of them fail to satisfy the buffer capacity constraints (e.g. a schedule is invalid if it requires a 4KB buffer, though the available buffer size is only 2KB.). Fig. 1 shows the performance distribution of the valid schedules. We observe a wide performance difference among the valid schedules, with the best one outperforming the worst one by $7.2\times$. In addition, we observe clusters of schedules that have similar latencies in the Fig. 1, revealing structure in the solution space.

B. State-of-the-art Schedulers

Given that the scheduling space for a DNN layer can have billions of valid schedules, finding a good schedule through exhaustive search can become an intractable problem. Table I shows some recent efforts to tackle this complexity.

1) Brute-force Approaches: Recent efforts combine exhaustive search with heuristics to manually prune the scheduling

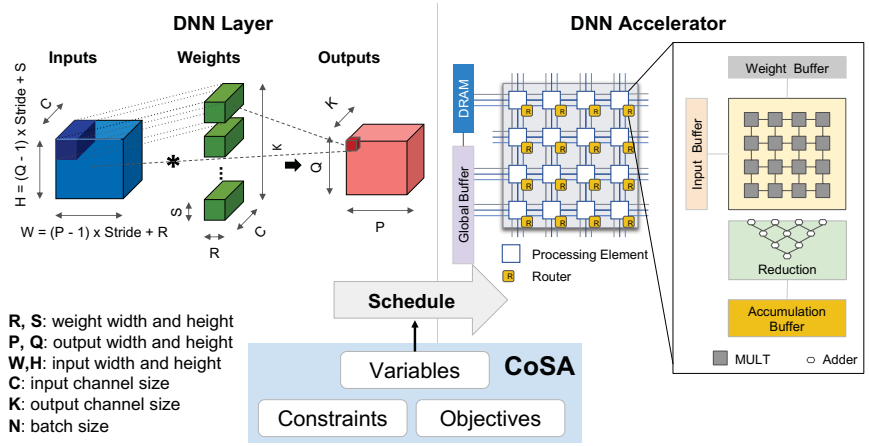


Fig. 2: DNN scheduling problem formulation with CoSA. CoSA takes 1) DNN layer dimensions and 2) DNN accelerator parameters and expresses the scheduling problem into a constrained optimization problem to produce a performant schedule in one shot.

```
1  //DRAM level
2  for q2 = [0 : 2) :
3   // Global Buffer level
4   for p2 = [0 : 7) :
5    for q1 = [0 : 7) :
6     for n0 = [0 : 3) :
7      spatial_for r0 = [0 : 3) :
8       spatial_for k1 = [0 : 2) :
9        // Input Buffer level
10       spatial_for k0 = [0 : 2) :
11        // Weight Buffer level
12        for c1 = [0 : 2) :
13         for p1 = [0 : 2) :
14          // Accumulation Buffer level
15          for s0 = [0 : 3) :
16           for p0 = [0 : 2) :
17            spatial_for c0 = [0 : 8) :
18             // Register
19              for q0 = [0 : 2) :
```

Listing 1: An example schedule using the loop nest representation for a DNN layer of dimension $R = S = 3, P = Q = 28, C = 8, K = 4, N = 3$. Same variable prefix indicates tiles from the same problem dimension.

space [14], [23], [49], [65], [71]. To lower the cost of exhaustive search, schedulers in this category typically use a lightweight analytical model to estimate latency, throughput, and power consumption to compare all valid mappings of a given layer to find the best schedule. The disadvantages of this approach are two-fold. First, such a brute-force search tends to be exceedingly expensive for complex hardware architectures, making it infeasible to find a good schedule quickly. Second, the generated schedules often do not perform optimally since analytical models may fail to consider the communication latency across the spatial hardware.

2) Feedback-based Approaches: Other recent efforts use feedback-driven approaches along with machine learning or other statistical methods [3], [15], [35], [38], [40], [56] to improve the accuracy of the cost model and search for the solution using black-box or gradient-based search. Although such approaches can potentially learn the distribution of the scheduling space, they typically require a large amount of training data due to their feedback-driven nature. As a result, these approaches are mainly applicable to post-silicon hardware where performing a large-scale measurement is possible but are not feasible for hardware under development.

3) Constrained-optimization Approaches: Constrained-optimization problems, in which objective functions are maximized or minimized subject to given sets of constraints, have demonstrated the ability to solve many complex large-scale problems in a reasonable time. Such methods have been widely used in architecture and systems research for instruction scheduling [20], [47], [48], high-level synthesis [22], memory partitioning [7], [37] [21], algorithm selection [33], [73], and program synthesis [4], [10], [52], [53], [62].

In particular, polyhedral transformation has leveraged constrained-optimization-based approach for auto-vectorization and loop tiling [2], [9], [13], [30], [42], [51]. Prior work targets general-purpose CPUs and GPUs that run with fine-grained instructions and hardware-managed cache, as opposed to the software-managed spatial accelerators that we target. In addition, existing polyhedral-based approaches [8], [9], [13] lack direct support for tile-size optimization. Instead, they take the tile size as input and apply a transformation based on the given tile size. Due to this limitation, the tile size decision cannot be co-optimized with other loop transformations, e.g. loop permutation, in one pass, leading to sub-optimal schedules.

To address the drawbacks of existing approaches and leverage the regularities from the DNN workloads and the accelerator design for optimization, CoSA employs constrained optimization to tackle the DNN scheduling problem in one pass. CoSA presents a unique domain-specific representation for DNN scheduling that better captures the utilization and communication cost and encodes different loop transformations, i.e., tiling size, loop permutation, and spatial mapping decisions, in one formulation. This unified representation enables us to solve for all three optimizations in one pass and produce efficient schedules for a complex accelerator system with a multi-level memory hierarchy.

III. The CoSA Framework

To navigate the large scheduling space of DNN accelerators, we develop CoSA, a constrained-optimization-based DNN scheduler to automatically generate high-performance schedules for spatially distributed accelerators. CoSA not only deterministically solves for a good schedule in one pass without the need for exhaustive search or iterative sampling, but can also be easily applied to different network layers and hardware architectures. This section discusses the CoSA framework and how CoSA formulates the DNN scheduling problem with mixed-integer programming (MIP).

A. CoSA Overview

CoSA optimizes operator-level schedules for mapping DNN layers onto spatial DNN accelerators. Specifically, CoSA formulates the scheduling problem as a constrained-optimization

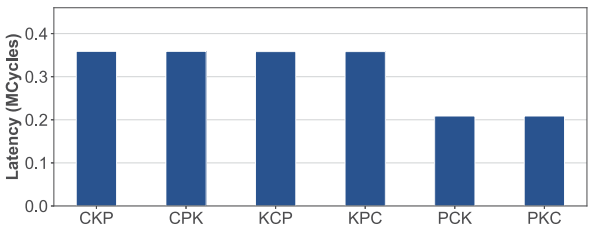


Fig. 3: Performance comparison of schedules with different loop permutations for a convolution operator with the layer dimensions of $R = S = 3$, $P = Q = 8$, $C = 32$, $K = 1024$. The leftmost schedule (`CKP`) refers to a relative ordering where the input channel dimension (`C`) is the outermost loop and the output height dimension (`P`) is the innermost loop. Since this layer is weight-heavy, loop permutations that emphasize weight reuse, e.g., `PCK` and `PKC`, are more efficient.

problem with *variables* representing the schedule, *constraints* representing DNN dimensions and hardware parameters, and *objective* functions representing goals, such as maximizing buffer utilization or achieving better parallelism. Fig. 2 shows the target problem space of CoSA. CoSA takes the specifications of the DNN layers and the underlying spatial accelerator as input constraints and generates a valid and high-performance schedule based on the objective functions in one pass.

1) Target Workload: The work targets the DNN operators that can be expressed by a nested loop with 7 variables as loop bounds: R, S, P, Q, C, K, N. R and S refer to the convolution kernel width and height, P and Q refer to the output width and height, C refers to the input channel size, K refers to the output channel size, and N refers to the batch size, as illustrated in Fig. 2. The convolution operation computes the dot product of the filter size $R \times S \times C$ of inputs and weights to generate one point in the output. Matrix multiplications can be expressed in this scheme as well.

2) Target Architecture: CoSA targets spatial architectures with an array of processing elements (PEs) connected via an on-chip network and with multiple levels of memory hierarchy, a commonly adopted architecture template in today's DNN accelerator designs [18], [19], [28], [29], [36], [44], [54], [55], [60], [69], [71].

3) Target Scheduling Decisions: CoSA-generated schedules describe how a specified DNN layer is executed on a given spatial architecture. Listing 1 shows an example of a schedule. Here, we use a loop-nest representation [49] to explicitly describe how the computation of a convolution layer is mapped to levels of memory hierarchies. We highlight three aspects of the schedule: 1) **loop tiling**, which describes which loops are mapped to which memory level and the values of the loop bounds; 2) **loop permutation**, which handles the relative ordering between loops in the same memory hierarchy; and 3) **spatial mapping**, which defines which loops are mapped to parallel spatial resources (shown as `spatial_for` loops in Listing 1). All three factors play a key role in the efficiency of the scheduling choice. Next, we highlight the implications of loop permutation and spatial mapping, both of which are less explored than the well-studied loop tiling.

Fig. 3 illustrates the impact of **loop permutation** for a

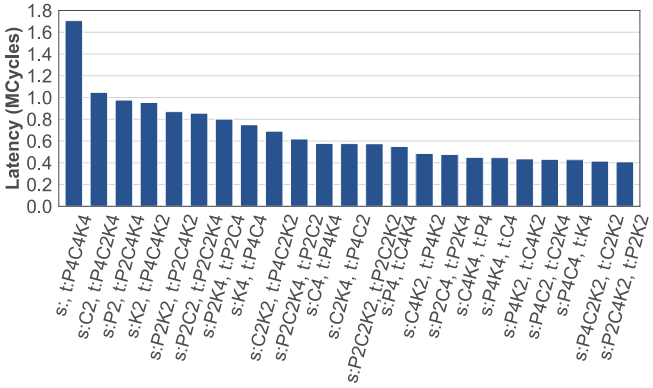


Fig. 4: Performance comparison of schedules with different spatial mappings for a convolution operator with the layer dimensions of $R = S = 1$, $P = Q = 16$, $C = 256$, $K = 1024$. Factors in s list are for spatial mapping, and factors in t list are for temporal mapping. For example, `s:P4C4,t:K4` represents a mapping where a factor 4 of the `P` dimension and a factor 4 of the `C` dimension are mapped to spatial execution in a system with 16 PEs, leaving `K`'s factor 4 to temporal mapping.

convolution layer on a given hardware design. All the schedules use the same loop tiling and spatial mapping except the loop ordering at the global-buffer level, as indicated in the labels of the X-axis, where `CKP` means the input channel dimension (`C`) is the outermost loop, and the output height dimension (`P`) is the innermost loop. In this case, selecting P as the outermost loop, i.e. `PCK` and `PKC`, can lead to a 1.7× speedup for this layer, motivating the need to consider the implications of loop permutation in the scheduling problem.

Fig. 4 shows the impact of **spatial mapping** on DNN execution. We notice that there is a 4.3× gap between best (rightmost) and worst (leftmost) schedules for the layer in consideration. The fundamental reason for the differences is the different communication traffic generated by different spatial mapping options. The best schedule, i.e., the rightmost schedule in the figure (`s:P2C4K2, t:P2K2`), is obtained when factors $P = 2$, $C = 4$, $K = 2$ are mapped to the spatial loops, which cannot be achieved by simply choosing either model or data parallelism in the spatial partition. As a result, a systematic evaluation of different spatial mapping choices is required to find a good schedule.

The rest of the section discusses how CoSA formulates the scheduling variables, constraints, and objectives to solve the DNN scheduling problem.

B. CoSA Variables and Constants

This section discusses the variables and constants, summarized in Table II, used in CoSA formulation.

CoSA Variables		CoSA Constants		Indices	
$\mathbf{X}$	binary matrix to represent a schedule	$\mathbf{A}$	layer dimension to data tensor mapping	i	memory level
		$\mathbf{B}$	memory level to data tensor mapping	j	layer dimension
				n	prime factor index
				k	mapping choice
				z	permutation level
				v	data tensor

TABLE II: CoSA Notations.

DNN Layer: $R = 3, S = 1, P = 1, Q = 1, C = 1, K = 4, N = 3$
$\rightarrow$ **Prime Factors:** = [[3],[1],[1],[1],[1],[2,2][3]]

Idx			Perm	Schedule								
j	Layer Dim.			R = 3		...	K = 4				N = 3	
n	Prime Factors			3		...	2		2		3	
k	s / t Mapping			s	t		s	t	s	t	s	t
i	Memory Levels	Register	...									
		...										
		InputBuf	...				✓					
		GlobalBuf	O_0									
			O_1									✓
			O_2						✓			
			...									
			O_Z	✓								

TABLE III: Example binary matrix $\mathbf{X}$ representing a schedule. A checkmark in s, t indicates spatial or temporal mapping. A checkmark in $O_0, ..., O_Z$ indicates the rank for loop permutation. In this schedule, the loop tile of size 3 from problem dimension N is allocated within the GlobalBuf at the innermost loop level, assigned for temporal execution. Both loop tiles from K are mapped to spatial resources.

1) Variable Representation: We devise a mathematical representation for the DNN schedules and formulate the scheduling problem as a prime-factor allocation problem. Given a layer specification, we first factorize each loop bound into its $prime\_factors$. If the loop bound themselves are large prime number, we can pad them and then factorize. We assign each prime factor to a *scheduling configuration* that is composed of a combination of three decisions: 1) the mapped memory level, 2) the permutation order, and 3) the spatial mapping. Each prime factor has exactly one scheduling configuration.

Here, we use a binary matrix $\mathbf{X}$ to represent the prime factor allocation, i.e., the scheduling space, shown in Table III. The four dimensions of $\mathbf{X}$ are: 1) the layer dimension variables (indexed by j), 2) the prime factors of the loop bounds (indexed by n), 3) whether it is a spatial or temporal mapping (indexed by k), and 4) the memory and the permutation levels (indexed by i). With the prime factor decomposition, CoSA's encoding can represent all possible schedules and guarantees that the optimization solves for the full search space.

Table III shows an example binary matrix X that represents the schedule shown in Listing 1. First, CoSA performs the *tiling* optimizations by assigning the prime factors to different memory levels. For example, dimension K is split into two tiles, where the inner tile of size 2 is allocated to the input buffer, and the outer tile of size 2 is allocated in the global buffer. Second, mapping a prime factor to *spatial* execution is indicated by whether the factor is mapped to a spatial column s or a temporal column t in the table. In this example, both prime factors for K are spatially mapped. Finally, for loop *permutation*, we add rank indices $O_0, O_1, ..., O_Z$ to the memory level of interest, where only one prime factor can be mapped to each rank. The lowest-ranked factor is allocated to the innermost loop, while the highest-ranked factor is allocated to the outermost loop. In the example shown in Table III, the problem dimension N is mapped at the O_1 level in the global buffer for temporal mapping, which means the factor $N = 3$ will be assigned rank 1 in the global-buffer level. Without other factors in the global-buffer level, factor $N = 3$ with the smallest rank will become the innermost loop in permutation. For the ranking of permutation, we reserve enough slots for all prime factors at all memory levels. Not all the slots need to be filled since a prime factor can only be allocated to one memory level.

	Related			Idx
	W	IA	OA	v
R	✓	-		j
S	✓	-		
P		✓	✓	
Q		✓	✓	
C	✓	✓		
K	✓		✓	
N		✓	✓	

	Related			Idx
	W	IA	OA	v
Register	✓	✓	✓	i
AccBuf			✓	
WBuf	✓			
InputBuf		✓		
GlobalBuf	✓	✓		
DRAM	✓	✓	✓	

TABLE IV: Constant binary matrices $\mathbf{A}$ (left) and $\mathbf{B}$ (right). $\mathbf{A}$ encodes how different layer dimensions associate with data tensors. $\mathbf{B}$ encodes which data tensor can be stored in which memory hierarchy.

2) Constant Parameters: In addition to the loop-related variables, we have intrinsic relations across different components in the architecture and layer specifications which must be encoded by constant parameters. CoSA uses two constant binary matrices to encode the unique relations in the DNN scheduling space, shown in Tabel IV. The first binary constant matrix, A, encodes the association between layer dimensions (i.e., rows of the matrix) and data tensors (i.e., columns of the matrix). For each input (IA), weight (W), and output (OA) tensor, matrix A indicates which layer dimensions, i.e., R, S, P, Q, C, K, N, should be used to calculate the data transaction size as well as multicast and reduction traffic on the accelerators.

In addition, we introduce another binary matrix $\mathbf{B}$ to represent which memory hierarchy can be used to store which data tensor. DNN accelerators typically deploy a multi-level memory hierarchy, where each memory level can be used to store different types of data tensors. For example, matrix $\mathbf{B}$ shown in Table IV represents an architecture that has dedicated input and weight buffers for input activation and weight, respectively, while providing a shared global buffer to store input and output activations.

C. CoSA Constraints

This section discusses the constraints derived from the target accelerator architecture that must be satisfied in CoSA and shows how to express them with CoSA variables and constants.

1) Buffer Capacity Constraint: To generate a valid schedule in a software-managed memory system, a key constraint is to ensure that the size of data to be sent to the buffer does not exceed the buffer capacity. The hardware memory hierarchy can be represented by the binary constant matrix $\mathbf{B}$ discussed earlier. For each memory buffer, based on the tensor-dimension correlation matrix $\mathbf{A}$, we calculate the tiling size of each tensor by multiplying the relevant prime factors together indicated by $\mathbf{X}$. Both spatial and temporal factors should be included in the buffer utilization. Let N_j be the number of prime factors for the layer dimension j. Then the utilization of the buffer level

I can be expressed as:

$$\prod_{i=0}^{I-1}\prod_{j=0,n=0}^{6,N_j}\prod_{k=0}^{1}\begin{cases}prime\_factor_{j,n}, & X_{(j,n),i,k}A_{j,v}B_{I,v}=1\\ 1, & \text{otherwise}\end{cases} \tag{1}$$

We then set the upper bound of the buffer utilization to the capacity of different buffer sizes, represented using $M_{I,v}$. However, a problem with this utilization constraint is that it involves products of the decision variables $\mathbf{X}$, making it nonlinear and infeasible to solve with standard constraint solvers. To address this limitation, we take the logarithm of both sides of the constraints to obtain a linear expression for the utilization and encode the if-else statement as:

$$\begin{aligned}U_{I,v} &= \sum_{i=0}^{I-1}\sum_{j=0,n=0}^{6,N_j}\sum_{k=0}^{1}\log(prime\_factor_{j,n})A_{j,v}B_{I,v}X_{(j,n),i,k}\\ &\leq \log(M_{I,v}),\forall I\end{aligned} \tag{2}$$

To encode different precisions for different data tensors, we add the logarithm of the datatype sizes $precision_v$ to $U_{I,v}$.

2) Spatial Resource Constraint: Another set of CoSA constraints is from the limited number of spatial resources. At the chip level, there is a limited number of PEs. At the PE level, there is a limited number of multiply-and-accumulate (MAC) units. In CoSA, once a factor is assigned to spatial mapping in the configuration, it needs to satisfy: 1) each problem factor can only be mapped to either spatial or temporal execution, 2) factors that map to spatial execution do not exceed the resource limit in the architecture. These two constraints can be expressed in the equations below:

$$\sum_{k=0}^{1}X_{(j,n),i,k} == 1,\forall(j,n),i \tag{3}$$

$$\sum_{j=0,n=0}^{6,N_j}\log(prime\_factor_{j,n})X_{(j,n),I,0}\leq\log(S_I),\forall I \tag{4}$$

where S_I is the number of available spatial resources at the level I.

D. Objective Functions

In this section, we describe the objective functions for CoSA. Each objective can be either used individually to optimize a single aspect of performance, e.g., utilization, compute, and communication, or combined with others.

1) Utilization-Driven Objective: High on-chip buffer utilization improves data-reuse opportunity. As demonstrated in the prior work [25], communication lower bounds can be achieved when the tiling block size is optimized for buffer utilization in a system with one-level cache. In this work, we formulate a utilization objective that aims to maximize the buffer utilization of all tensors, so the overall communication is minimized. We use the same formulation for the buffer utilization as in III-C1 and maximize the following linear utilization function:

$$\hat{Util} = \sum_{i=0}^{I-1}\sum_{v=0}^{2}U_{i,v} \tag{5}$$

Here, maximizing the sum of utilization for all buffer levels and all tensors in the logarithm form is equivalent to maximizing the geometric mean of the buffer utilization. Users can also attach weights to the different buffer levels or different data tensors if they want to optimize for the utilization of a specific level of the memory.

2) Compute-Driven Objective: The total number of compute cycles is another factor that affects the quality of schedules. In this formulation, we multiply all the temporal factors for the estimated compute cycles in each PE. Intuitively, this objective allows the constraint solver to exploit the parallelism in the system by mapping more iterations to the spatial resources than to temporal iterations. The objective can be expressed as a linear function again with logarithm taken:

$$\hat{Comp} = \sum_{i=0}^{I}\sum_{j=0,n=0}^{6,N_j}\log(prime\_factor_{j,n})X_{(j,n),i,1} \tag{6}$$

3) Traffic-Driven Objective: Communication latency is a key contributing factor to the performance of spatial architecture. CoSA also includes a traffic-driven objective to capture the communication cost. Specifically, communication traffic can be decomposed into three terms: 1) data size per transfer, 2) spatial factors of multicast and unicast traffic, and 3) temporal iterations. Multiplying these three factors will get the total amount of traffic in the network. Next, we discuss how we capture each of these factors using CoSA's representation.

First, similar to the buffer utilization expression, data size per transfer can computed using the allocated prime factors in matrix $\mathbf{X}$, together with the dimension-tensor correlation matrix $\mathbf{A}$, as shown in the equation below:

$$D_v = \sum_{i=0}^{I-1}\sum_{j=0,n=0}^{6,N_j}\sum_{k=0}^{1}\log(prime\_factor_{j,n})A_{j,v}X_{(j,n),i,k} \tag{7}$$

Second, spatial factors would incur different multicast, unicast, and reduction patterns. The dimension-tensor correlation matrix $\mathbf{A}$ discussed in Sec III-B2 can be used to indicate different traffic patters. Specifically, depending on whether the spatial dimension, indicated by the binary matrix $\mathbf{X}$, is related to the specific tensor in consideration, represented by the constant matrix $\mathbf{A}$, different traffic patterns, e.g., multicast vs. unicast or reduction vs. unicast, would occur.

Fig. 5 shows how the intrinsic tensor-dimension correlation matrix $\mathbf{A}$ can be used to calculate different traffic patterns for different variables. For example, as shown in Fig. 5a, if the dimension P is mapped spatially, $A_{P,\mathrm{W}}=0$ implies multicast traffic for weight tensor W. Since weight is not related to P, when we send weights from global buffer to PEs, the weight traffic will be multicasted to the destination PEs. If the dimension C is mapped spatially, $A_{C,\mathrm{W}}=1$ (Fig. 5b) implies unicast traffic for weight tensor W as weight is related to C. Similarly, if the dimension C is mapped spatially, $A_{C,\mathrm{OA}}=0$ (Fig. 5c) implies reduction traffic for output tensor OA, where partially sum needs to be reduced across C before sending back to GB. If the dimension P is mapped spatially, $A_{P,\mathrm{OA}}=1$

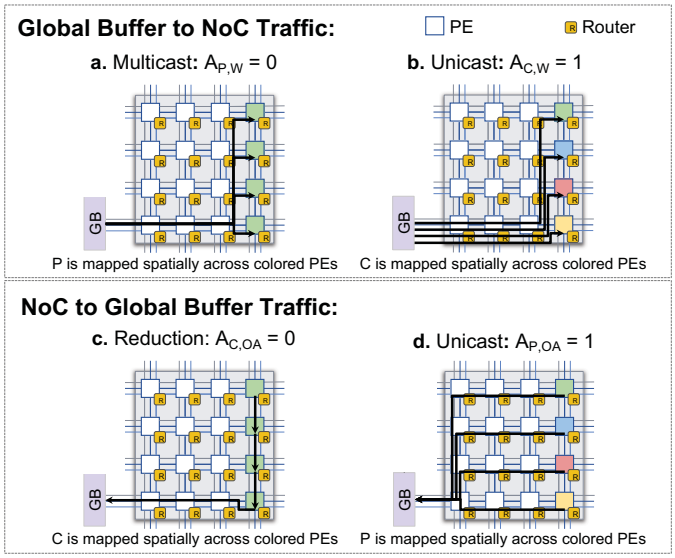


Fig. 5: Different traffic patterns based on the constant matrix $\mathbf{A}$. The two figures (top) show how the constant $\mathbf{A}$ encodes the traffic types (multicast, unicast, reduction) for different data tensors from the global buffer to PEs. The figures on the bottom show its implication on output tensor reduction traffics.

(Fig. 5d) would indicate unicast traffic for output tensor OA, as each traffic contributes to different regions of the output. CoSA formulates this relationship in the following equation:

$$L_v = \sum_{j=0,n=0}^{6,\,N_j} \log(prime\_factor_{j,n}) X_{(j,n),I,0} A_{j,v} \tag{8}$$

The third term, temporal iteration is used to calculate the number of data transfers at the NoC level. We introduce a traffic iteration factor $\mathbf{Y}$ that is a function of $\mathbf{X}$ at the permutation level, $\mathbf{A}$, and $\mathbf{B}$. $\mathbf{Y}$ indicates if the outer NoC loop bound should be used for different variables. With $\mathbf{Y}$, we ensure that, for each variable, if a relevant factor term is seen inside the current loop level, the current loop level's factor should be used to compute the traffic iteration regardless of whether it is related to the data tensor of the variable of interest. This is a term that drives the reuse optimization. Mathematically, $\mathbf{Y}$ is constrained as:

$$\begin{aligned} Y_{v,z} &\geq \sum_{j=0,n=0}^{6,\,N_j} X_{(j,n),z,1} A_{j,v} B_{I,v}, \forall z, \forall v \\ Y_{v,z} &\geq Y_{v,z-1}, \forall z > 0, \forall v \end{aligned} \tag{9}$$

Where z represents the position index for permutation and Z equals the total valid levels for permutation. The traffic iteration term can thus be expressed as:

$$T_v = \sum_{z=0}^{Z-1} \sum_{j=0,n=0}^{6,N_j} \log(prime\_factor_{j,n}) Y_{v,z} X_{(j,n),z,1} \tag{10}$$

This turns the linear objective into quadratic as we multiply $\mathbf{Y}$ with $\mathbf{X}$ to indicate whether there is a factor at the current permutation level.

After we calculate each individual term, we can combine them together for each tensor that contributes to the total traffic in the network. Similar to the logarithmic transformation we did earlier, instead of multiplying these three terms together, we take the logarithm on both sides to get a linear expression of the traffic, as shown in the equation below:

$$\hat{Traf} = \sum_{v=0}^{2} (D_v + L_v + T_v) \tag{11}$$

4) Overall Objective: One can construct a composite objective comprised of a linear combination of $\hat{Util}$, $\hat{Comp}$, and $\hat{Traf}$, where we want to minimize the compute and communication latency while maximizing the on-chip buffer utilization:

$$\hat{O} = -w_U \hat{Util} + w_C \hat{Comp} + w_T \hat{Traf} \tag{12}$$

where w_U, w_T, w_C are user-selected parameters controlling the importance of each objective. For a system with double-buffering optimization, w_T can be set to map the traffic sizes to the cycles for memory accesses. This brings $w_T \hat{Traf}$ to be of the same importance as $w_C \hat{Comp}$ in the optimization. Another formulation of the overall objective function to balance the memory access and compute cycles is to minimize the difference of the two terms: $\hat{D} = w_T \hat{Traf} - w_C \hat{Comp}$. The weights of different objectives can be determined by using a set of micro-benchmarks that characterize the compute, memory, and communication latencies of the target architecture.

E. Limitation of CoSA

CoSA leverages the regularity from both the problem and the architecture space, where it assumes a dense CNN workload and does not exploit the sparsity of the data. It also best targets hardware systems with deterministic behavior and explicitly managed scratchpads. This is because, in systems with non-deterministic behaviors, it can be challenging to construct optimization objectives that capture the impact of such behaviors. However, CoSA can be augmented with an iterative search on the objective functions and their corresponding hyperparameters to approximate the unknown hardware performance model and directly prune off the invalid points from the search space.

IV. Methodology

This section discusses the evaluation platforms we use followed by the experimental setup for CoSA evaluation.

A. Evaluation Platforms

We evaluate the schedules generated by CoSA on two platforms: 1) Timeloop for cycle performance and energy consumption, and 2) our cycle-exact NoC simulator for overall latency performance. The latter more accurately captures the communication overhead and concurrent hardware behaviors on a spatial architecture.

Timeloop provides microarchitecture and technology-specific energy models for estimating the performance and energy on DNN accelerators. Timeloop reports the performance in terms of the maximum cycles required for each processing element to complete the workload and to perform

Arithmetic :		*Storage :*		*Network :*	
MACs	64 / PE	**Registers**	64B / PE	**Dimension**	4×4
Weight/Input Precision	8bit	**Accum. Buffer**	3KB / PE	**Router**	Wormhole
		Weight Buffer	32KB / PE	**Flit Size**	64b
Partial-Sum Precision	24bit	**Input Buffer**	8KB / PE	**Routing**	X-Y
		Global Buffer	128KB	**Multicast**	Yes

TABLE V: The baseline DNN accelerator architecture.

memory accesses, assuming perfect latency hiding with double buffering. The energy consumption in Timeloop is calculated by multiplying the access count on each hardware component with the energy per access and summing the products up. The access count is inferred from the schedule and the energy per access is provided by an energy reference table in Timeloop.

NoC Simulator augments the Timeloop analytical compute model for PEs with a synthesizable NoC implementation to reflect the communication cost. Communication is one of the key contributing factors for latency in a NoC-based system, especially for the communication bound schedules.

The NoC simulator is transaction-based and cycle-exact for modeling the on-chip traffic. Leveraging the synthesizable SystemC router design from Matchlib [41] that supports unicast and multicast requests, we construct a resizable 2-D mesh network and implement an X-Y routing scheme. The simulator captures both computation and communication latencies by concurrently modeling data transfers in the NoC, the PE executions, and off-chip DRAM accesses based on the DRAMSim2 model [58], where the impact of traffic congestion on the NoC can also be manifested.

B. Baseline Schedulers

We evaluate CoSA with respect to two other scheduling schemes: 1) a **Random** scheduler that searches for five different valid schedules, from which we choose the one with the best result for the target metric, and 2) the **Timeloop Hybrid** mapper in Timeloop [49] that randomly selects a tiling factorization, prunes superfluous permutations, and then linearly explores the pruned subspace of mappings before it proceeds to the next random factorization. For this mapper, we keep the default termination condition where each thread self-terminates after visiting 500 consecutive mappings that are valid yet sub-optimal. The mapper is run with 32 threads, each of which independently searches the scheduling space until its termination condition is met. Once all threads have terminated, Timeloop returns the best schedule obtained from all 16,000+ valid schedules.

C. Experiment Setup

Mixed-Integer Program (MIP) Solver: CoSA uses Gurobi [32], a general-purpose optimization solver for MIP and other constrained programming, as the solver. We specify the CoSA variables, constraints, and objective functions before we invoke the solver. The solver takes at most seconds to return a schedule for DNN layers.

DNN workloads: We measure the performance of CoSA-generated schedules over a wide range of DNN workloads targeting different DNN tasks with diverse layer dimensions, including: ResNet-50 [34], ResNeXt-50 (32x4d) [70], and Deepbench [24] (OCR and Face Recognition). The precision used for the benchmarks is 8-bit for the input and weights, and 24-bit for the partial sums. We do not pad the dimensions to be multiples of 2, as it incurs more overhead and outweighs the benefits it provides to allow more scheduling options.

Baseline architecture: We consider a spatial-array architecture like Simba [59] as our baseline. Detailed specifications of the hardware constructs are summarized in Table V. We demonstrate that the CoSA framework is general to be applied for different architecture parameters while delivering high-performance scheduling options in one shot.

V. Evaluation

In this section, we demonstrate the improved time-to-solution, performance, and energy of CoSA compared to baseline schedulers, across different evaluation platforms and different DNN architectures on a diverse set of DNN layers.

A. Time to Solution

We compare the average time for CoSA and the baseline schedulers to generate the schedule of each layer from the four target DNN workloads. Table VI shows that CoSA's optimization-driven approach offers more than 90× (4.2s vs. 379.9s) time-to-solution advantage over the Timeloop Hybrid search strategy. Timeloop Hybrid search sampled 67 million schedules per layer and evaluated more than 16 thousand valid ones among them, leading to a long runtime. With Random search, a random sampling of 20K samples in 4.6 seconds resulted in only five valid schedules, further demonstrating the need to have a constraint-based strategy to prune the invalid search space directly. In the following section, we show that CoSA not only shortens the time-to-solution but also generates high-quality schedules.

	CoSA	Random (5×)	Timeloop Hybrid
Avg. Runtime / Layer	**4.2s**	4.6s	379.9s
Avg. Samples / Layer	**1**	20K	67M
Avg. Evaluations / Layer	**1**	5	16K+

TABLE VI: Time-to-solution Comparison. CoSA outputs only one valid schedule per layer. CoSA's runtime is 1.1× and 90× shorter than the Random and Timeloop Hybrid search, respectively.

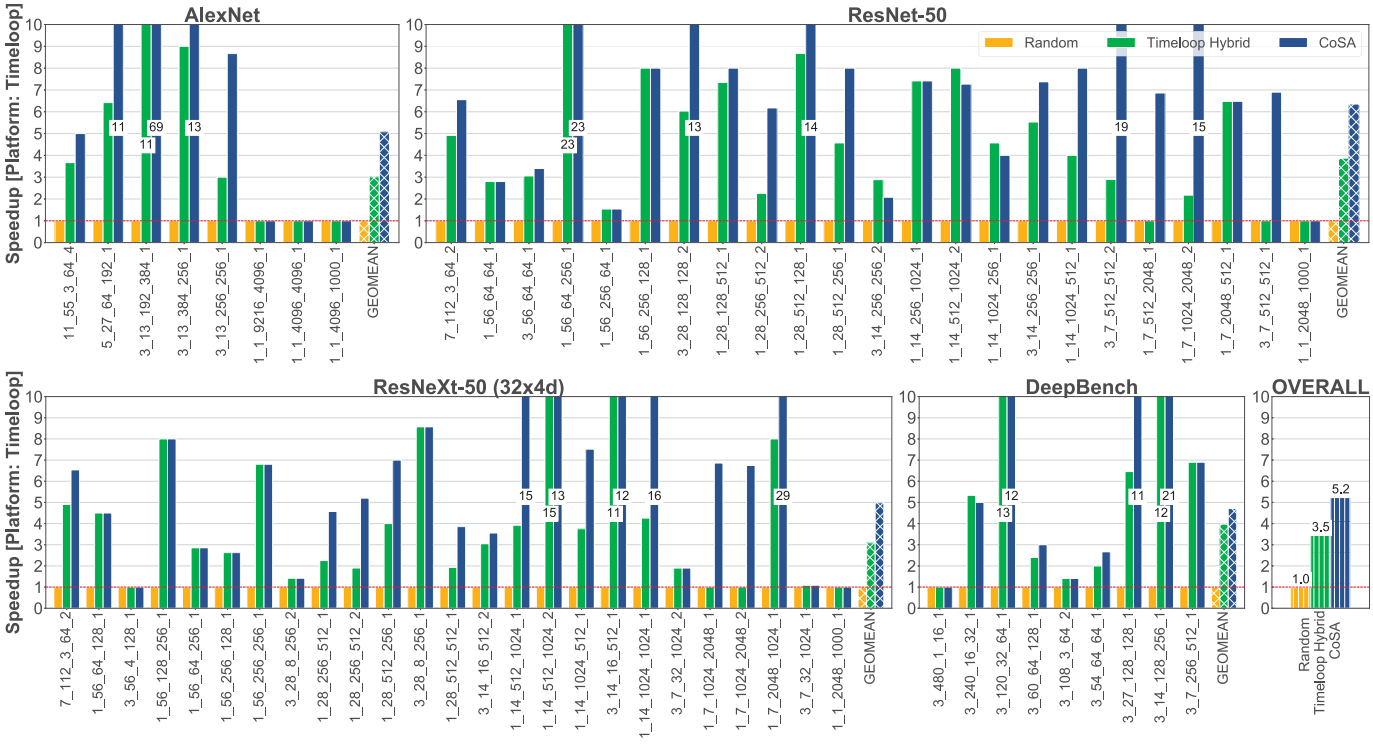


Fig. 6: Speedup of different schedules relative to Random search on the baseline 4×4 NoC architecture. X-axis labels follow the naming convention `R_P_C_K_Stride` where $S = R$ and $Q = P$ in all workloads. CoSA achieves 5.2× and 1.5× higher geomean speedup across four DNN workloads compared to the Random and Timeloop Hybrid search.

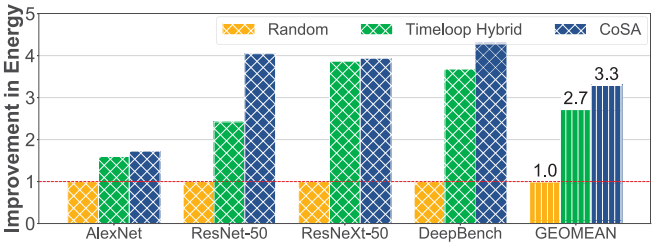


Fig. 7: Improvements in total network energy reported by the Timeloop energy model. Energy estimations are normalized to results from Random search and are evaluated on the baseline 4×4 NoC.

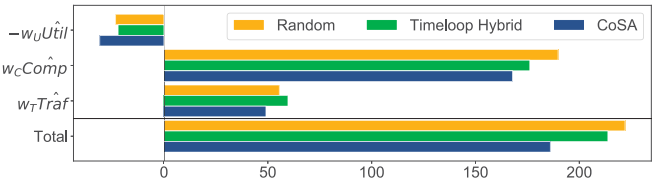


Fig. 8: Objective function breakdown for ResNet-50 layer 3_7_512_512_1. The goal is to minimize the total objective in Eq. 12. CoSA achieves the lowest values for all objective functions on this layer among all approaches.

B. Evaluation on Timeloop Performance and Energy Models

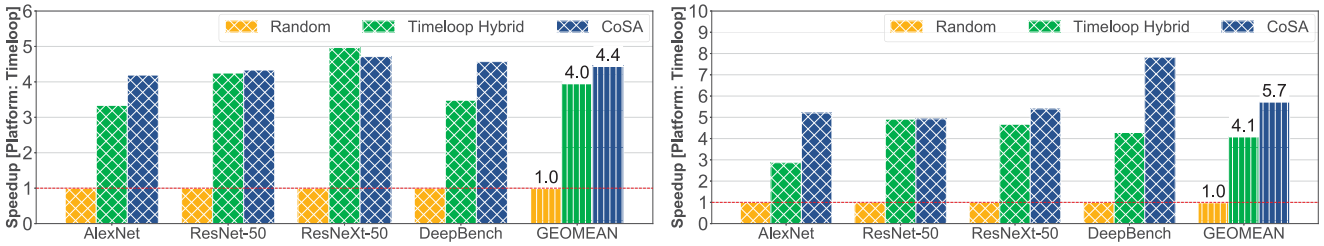
We compare the performance of the Random search, the Timeloop Hybrid mapper, and the CoSA scheduler for four different DNN workloads. The evaluations are based on our baseline architecture described in Table V and the Timeloop evaluation platform mentioned in Section IV-A.

1) Performance: Fig. 6 shows the speedup reported by Timeloop for different scheduling schemes relative to Random search. Fig. 6 demonstrates that the CoSA-generated schedules are not only valid but also outperform the ones generated by both Random search and Timeloop Hybrid search. The geometric mean of the speedups of CoSA schedules relative to the Random and Timeloop Hybrid search ones are 5.2× and 1.5× respectively across four DNNs.

In the few layers where Timeloop Hybrid search slightly outperforms CoSA, we find a higher iteration count at the DRAM level in Timeloop Hybrid schedules, which helps to reduce the size of each DRAM transaction and balance the pipeline. Fine tuning the weights of the objective functions could be used to further improve the CoSA-generated schedules.

A more exhaustive Timeloop Hybrid search (32K valid schedules) results in an improvement of only 7.5% in latency while increasing runtime by 2×. We find that even with 2× more valid samples evaluated, Timeloop Hybrid search still cannot generate schedules that are of similar efficiency to CoSA.

2) Energy: We use the Timeloop energy model to evaluate the energy of different schedules. Because energy cost is highly correlated with the access count on each hardware component, our traffic objective in CoSA is used for the schedule optimization targeting energy efficiency. Fig. 7 demonstrates that CoSA, using no simulation feedback, can generate schedules 22% more energy-efficient than the best Timeloop Hybrid solutions



(a) 8×8 PEs (b) Larger Buffers

Fig. 9: Speedup relative to Random search reported by Timeloop model on different hardware architectures. CoSA's performance generalizes across different hardware architectures with different computing and on-chip storage resources.

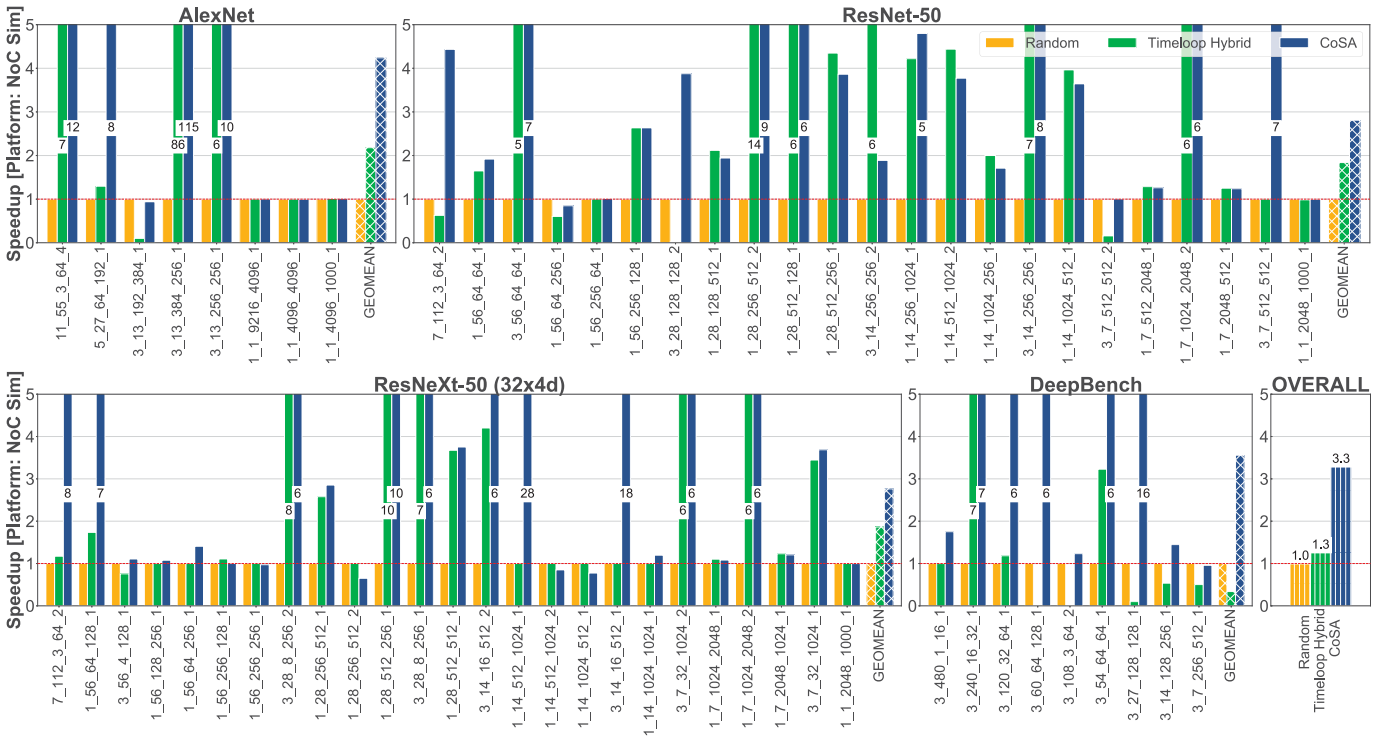


Fig. 10: Speedup reported by NoC simulator relative to Random search on the baseline 4×4 NoC architecture. CoSA achieves 3.3× and 2.5× higher geomean speedup across four DNN workloads compared to the Random and Timeloop Hybrid search on the more communication sensitive NoC simulator.

selected from 16,000+ valid schedules optimizing the energy.

3) Objective Breakdown: A detailed breakdown of the CoSA objective function on ResNet50 layer 3_7_512_512_1 is included in Fig.8. Our overall objective function aims to capture an optimization heuristic to maximize the utilization and minimize the compute and traffic costs at the same time with a weighted sum of the three. Fig.8 shows that CoSA achieves the lowest total objective among all approaches, and optimizes all three sub-objectives simultaneously. This observation on the objective values aligns with our empirical results in Fig. 6, where CoSA schedule runs 7× faster than the ones generated by Random and Timeloop Hybrid search.

4) Different HW Architectures: We further explore the performance of CoSA with different DNN architecture parameters such as different PE array sizes and different SRAM buffer sizes. We apply the same weights for the evaluation on the same architecture and customize the objective weights in Eqn.12 using a micro-benchmark for different architectures. Fig.9 shows the geomean speedup of CoSA across all networks on two different hardware architectures.

PE Array Dimension. We scale the number of PEs up by 4× and increase both the on-chip communication and DRAM bandwidth by 2× correspondingly. Both of these modifications significantly impact the compute and communication patterns of DNN layer executions. With a larger spatial array of arithmetic units, this case study presents a scheduling problem where decisions about spatial and temporal mapping can be especially crucial to attaining high performance. Fig. 9a shows that CoSA achieves 4.4× and 1.1× speedup compared to Random and Timeloop Hybrid search respectively across four networks. This shows that the performance of our scheduler can scale and generalize to NoCs with more PEs, which tend to be more

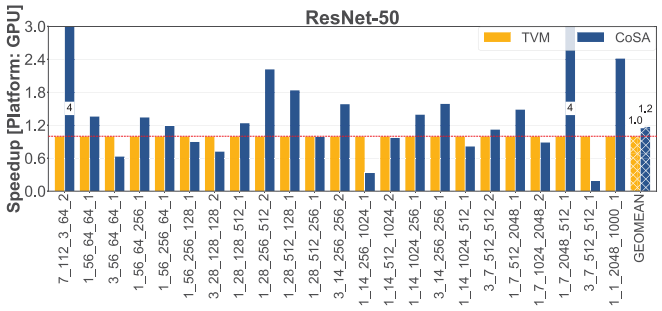


Fig. 11: Speedup relative to TVM reported on K80 GPU.

affected by communication costs.

SRAM Size. We also increase the sizes of the local and global buffers to demonstrate that CoSA can achieve consistently good schedules across different architectures. The sizes of local buffers, i.e. accumulation, weight, and input buffers, are doubled and the global buffer size increased $8\times$. Modified memory capacities, at the PE and global buffer level, are likely to impact the optimal strategy for data re-use and NoC communication traffic reduction. With CoSA, we show $5.7\times$ speedup over Random and $1.4\times$ speedup over Timeloop Hybrid search in Fig. 9b, demonstrating CoSA's capability across different architectures.

C. Evaluation on NoC Simulator

To further compare the quality of schedules generated by different scheduling schemes, we evaluate them on our NoC simulation platform. The NoC simulation platform more accurately captures the communication overhead from the on-chip network as compared to the Timeloop models.

Fig. 10 shows the speedup relative to the Random baseline. We observe that CoSA-generated schedules outperform the baseline schedules for all four DNN workloads, with the greatest performance gains occurring for convolutional layers, e.g. DeepBench layers. Intriguingly, for these same layers, Timeloop Hybrid scheduler actually under-performs Random search as its internal analytical model does not accurately capture the communication traffic in the network. On the other hand, there is no significant difference between the performance of FC layers among different schedules, as the FC layers are heavily memory-bound with low PE utilization. The DRAM access time dominates in these layers even with the best schedules with respect to reuse of buffered data.

Overall, CoSA achieves a geometric average of up to $3.3\times$ speedup relative to the best Random search solutions and $2.5\times$ relative to Timeloop Hybrid search schedules across the four networks. Furthermore, unlike the iterative nature of Random and Timeloop Hybrid search schedules, CoSA schedules are consistently performant with the one-shot solution.

D. Evaluation on GPU

To show the potential use of CoSA for general-purpose hardware, we also formulate GPU scheduling as a constrained-optimization problem using CoSA. We evaluate the performance of CoSA on GPU and compare it against TVM [15].

Target GPU. We target NVIDIA K80 GPU with 2496 CUDA cores and a 1.5MB L2 cache. This GPU has a 48KB shared memory and 64KB local registers, shared by a maximum of 1024 threads in each CUDA thread block. The thread block is a programming abstraction that represents a group of threads that can be run serially or in parallel in CUDA. The maximum dimension of a thread block is (1024, 1024, 64). Violating these constraints in the CUDA kernel results in invalid schedules.

Constraints. CoSA expresses the hardware constraints for GPU thread groups and shared/local memory similarly to how we specify the spatial resource and buffer capacity constraints in Section III-C. Each thread group can be seen as a spatial level with a specific size. The product of all three thread group sizes is enforced to be smaller than 1024. The share memory utilization is calculated as buffer capacity constraints, and the register utilization is calculated by multiplying the total number of threads with the inner loop register utilization.

Objective Functions. In CoSA, we compute the compute objective by discounting the total compute cycles with the total number of threads for GPU, to reflect the performance gain from thread-level parallelism. We then adjust the weights of the other objectives using a micro-benchmark.

We run TVM with the XGBoost tuner for 50 trials per layer as the baseline. CoSA generates valid schedules in one shot with a time-to-solution $2,500\times$ shorter than TVM (0.02s vs. 50s per layer). The CoSA-generated schedules achieve $1.10\times$ geomean speedup compared to the TVM schedules on ResNet50 as shown in Fig.11.

VI. Conclusion

In this paper, we present CoSA, an optimization-driven approach to DNN scheduling. Harnessing the regularities from DNN workloads and target accelerator designs, we formulate scheduling into a constrained optimization problem that can be solved directly without incurring the high cost of iterative scheduling. We devise a single mathematical formulation to simultaneously solve for all three key optimizations in scheduling: loop tiling, loop permutation, and spatial mapping. Comparing our results to schedules generated from the state-of-the-art work, our approach achieves up to $2.5\times$ speedup and 22% better energy-efficiency, with $90\times$ shorter time-to-solution.

Acknowledgements

The authors would like to thank Lianmin Zheng for providing the TVM tuning scripts and scheduling templates, and Kostadin Ilov for the computing system support. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, Berkeley Wireless Research Center, ADEPT Lab industrial sponsors (Intel, Apple, Futurewei, Google, Qualcomm, Seagate, Western Digital), and a Facebook Faculty Research Award.

REFERENCES

[1] "Edge TPU," https://cloud.google.com/edge-tpu/, accessed: 2018-12-05.

[2] A. Acharya, U. Bondhugula, and A. Cohen, "Polyhedral auto-transformation with no integer linear programming," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[3] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, and J. Ragan-Kelley, "Learning to optimize halide with tree search and random programs," *ACM Transactions on Graphics (TOG)*, 2019.

[4] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, "Search-based program synthesis," *Communications of the ACM*, 2018.

[5] Amazon, "AWS Inferentia: High Performance Machine Learning Inference Chip," https://aws.amazon.com/machine-learning/inferentia/, 2018.

[6] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[7] O. Avissar, R. Barua, and D. Stewart, "Heterogeneous memory management for embedded systems," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2001.

[8] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *International Symposium on Code Generation and Optimization (CGO)*, 2019.

[9] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. Van Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev, "Pencil: A platform-neutral compute intermediate language for accelerator programming," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2015.

[10] S. Bansal and A. Aiken, "Automatic generation of peephole superoptimizers," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2006.

[11] M. Bojarski, P. Yeres, A. Choromanska, K. Choromanski, B. Firner, L. Jackel, and U. Muller, "Explaining how a deep neural network trained with end-to-end learning steers a car," 2017.

[12] U. Bondhugula, A. Acharya, and A. Cohen, "The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2016.

[13] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[14] P. Chatarasi, H. Kwon, N. Raina, S. Malik, V. Haridas, A. Parashar, M. Pellauer, T. Krishna, and V. Sarkar, "Marvel: A data-centric compiler for dnn operators on spatial accelerators," 2020.

[15] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An Automated End-to-end Optimizing Compiler for Deep Learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[16] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, March 2014.

[17] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[18] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.

[19] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A Machine-learning Supercomputer," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014.

[20] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to cgra mapping," in *Design Automation Conference (DAC)*, 2018.

[21] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2011.

[22] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," in *Design Automation Conference (DAC)*, 2006.

[23] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shrivastava, "DMazeRunner: Executing perfectly nested loops on dataflow accelerators," *ACM Transactions on Embedded Computing Systems*, 2019.

[24] DeepBench, "http://www.github.com/baidu-research/deepbench."

[25] G. Dinh and J. Demmel, "Communication-optimal tilings for projective nested loops with arbitrary bounds," 2020.

[26] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[27] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. Reinhardt, A. Caulfield, E. Chung, and D. Burger, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018.

[28] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2017.

[29] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, "Tangram: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2019.

[30] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in llvm," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2011.

[31] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The architectural implications of facebook's dnn-based personalized recommendation," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[32] L. Gurobi Optimization, "Gurobi optimizer reference manual," 2020. [Online]. Available: http://www.gurobi.com

[33] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, Shih-Wei Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the suif compiler," *IEEE Computer*, 1996.

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[35] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, "Mind mappings: enabling efficient algorithm-accelerator mapping space search," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2021.

[36] G. Henry, P. Palangpour, M. Thomson, J. S. Gardner, B. Arden, J. Donahue, K. Houck, J. Johnson, K. O'Brien, S. Petersen, B. Seroussi, and T. Walker, "High-performance deep-learning coprocessor integrated into x86 soc with server-class cpus industrial product," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020.

[37] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella, "Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2020.

[38] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks." in *Proceedings of Machine Learning and Systems (MLSys)*, 2019.

[39] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski,

A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[40] S.-C. Kao and T. Krishna, "GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2020.

[41] B. Khailany, E. Krimer, R. Venkatesan, J. Clemons, J. S. Emer, M. Fojtik, A. Klinefelter, M. Pellauer, N. Pinckney, Y. S. Shao, S. Srinath, C. Torng, S. L. Xi, Y. Zhang, and B. Zimmer, "Invited: A modular digital vlsi flow for high-productivity soc design," in *Design Automation Conference (DAC)*, 2018.

[42] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet simd code generation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," in *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2012.

[44] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Programmable Interconnects," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2018.

[45] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Transactions on Graphics (TOG)*, 2016.

[46] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep learning recommendation model for personalization and recommendation systems," 2019.

[47] T. Nowatzki, N. Ardalani, K. Sankaralingam, and J. Weng, "Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2018.

[48] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," *ACM SIGPLAN Notices*, 2013.

[49] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A Systematic Approach to DNN Accelerator Evaluation," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.

[50] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[51] E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan, "Predictive modeling in a polyhedral optimization space," *International journal of parallel programming*, 2013.

[52] P. M. Phothilimthana, A. S. Elliott, A. Wang, A. Jangda, B. Hagedorn, H. Barthels, S. J. Kaufman, V. Grover, E. Torlak, and R. Bodik, "Swizzle inventor: Data movement synthesis for gpu kernels," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2019.

[53] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik, "Chlorophyll: Synthesis-aided compiler for low-power spatial architectures," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[54] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," 2017.

[55] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[56] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, 2013.

[57] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-time Object Detection," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[58] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE computer architecture letters*, 2011.

[59] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019.

[60] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019.

[61] F. Sijstermans, "The NVIDIA Deep Learning Accelerator," in *Hot Chips*, 2018.

[62] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[63] J. Song, Y. Cho, J.-S. Park, J.-W. Jang, S. Leev, J.-H. Song, J.-G. Lee, and I. Kang, "An 11.5 tops/w 1024-mac butterfly structure dual-core sparsity-aware neural processing unit in 8nm flagship mobile soc," in *Proceedings of the International Solid State Circuits Conference (ISSCC)*, 2019.

[64] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2014.

[65] P. Tillet, H. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019.

[66] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," 2016.

[67] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," 2018.

[68] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[69] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[70] S. Xie, R. Girshick, P. Dollar, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[71] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2020.

[72] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An Accelerator for Sparse Neural Networks," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2016.

[73] R. Zhou and T. M. Jones, "Janus: Statically-driven and profile-guided automatic dynamic binary parallelisation," in *International Symposium on Code Generation and Optimization (CGO)*, 2019.