2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2027-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPCA53966.2022.00042

Atomic Dataflow based Graph-Level Workload Orchestration for Scalable DNN Accelerators

Shixuan Zheng, Xianjue Zhang, Leibo Liu, Shaojun Wei, Shouyi Yin*
Tsinghua University, Beijing, China
*Corresponding author: yinsy@tsinghua.edu.cn

***Abstract*—To efficiently deploy state-of-the-art deep neural network (DNN) workloads with growing computational intensity and structural complexity, scalable DNN accelerators have been proposed in recent years, which are featured by multi-tensor engines and distributed on-chip buffers. Such spatial architectures have significantly expanded scheduling space in terms of parallelism and data reuse potentials, which demands for delicate workload orchestration. Previous works on DNN's hardware mapping problem mainly focus on operator-level loop transformation for single array, which are insufficient for this new challenge. Resource partitioning methods for multi-engines such as CNN-partition and inter-layer pipelining have been studied. However, their intrinsic disadvantages of workload unbalance and pipeline delay still prevent scalable accelerators from releasing full potentials.**

In this paper, we propose *atomic dataflow*, a novel graph-level scheduling and mapping approach developed for DNN inference. Instead of partitioning hardware resources into fixed regions and binding each DNN layer to a certain region sequentially, atomic dataflow schedules the DNN computation graph in workload-specific granularity (atoms) to ensure PE-array utilization, supports flexible atom ordering to exploit parallelism, and orchestrates atom-engine mapping to optimize data reuse between spatially connected tensor engines. Firstly, we propose a simulated annealing based atomic tensor generation algorithm to minimize load unbalance. Secondly, we develop a dynamic programming based atomic DAG scheduling algorithm to systematically explore massive ordering potentials. Finally, to facilitate data locality and reduce expensive off-chip memory access, we present mapping and buffering strategies to efficiently utilize distributed on-chip storage. With an automated optimization framework being established, experimental results show significant improvements over baseline approaches in terms of performance, hardware utilization, and energy consumption.

***Keywords*-scheduling; domain-specific architectures;**

I. Introduction

Ranging from edge applications to cloud computing scenarios, domain-specific DNN accelerators have become prevalent in the past decade. Benefited from customized data-path, local reuse-oriented memory hierarchy, and regular processing element (PE) arrays with delicate dataflow mechanism, spatial architectures have shown dramatic enhancement over general-purpose processors in terms of performance and energy efficiency [1], [13], [17], [25], [36], [40], [44].

DNN workloads continuously evolve in both computational scale and structural complexity, such as the newly proposed image classifiers with hundreds of millions of parameters [22], [23], [48] and neural architecture search (NAS) generated networks which are wired in irregular topology [39], [64]. Google reports a $\sim$1.5$\times$ annual growth of both memory and computational cost of TPU's workloads. Hence, there is strong motivation of developing scalable DNN architectures to deploy large-scale complex DNN models efficiently.

Recently, scalable DNN accelerators are proposed to support high performance deep learning applications [55]. Neurocube [28] and TETRIS [18] integrate multiple logic dies with vertical storage chips to overcome memory bottleneck, and sequentially partition each DNN layer to exploit intra-layer parallelism. Tangram [19] proposes inter-layer pipelining across multi-engines to reduce off-chip memory access. HDA [33] combines heterogeneous accelerators to deploy multi-tenancy workloads.

Dataflow design of DNN workloads determines to what extend the potential of spatial architectures can be reached. It includes both tensor computation scheduling and task-hardware mapping. Extensive previous works have been done to optimize fine-grained dataflow for spatial 2D PE arrays via loop transformation techniques (e.g., reorder, unroll, split, fuse) [32], [37], [59], [63]. However, since scalable accelerators have enabled parallel task execution among multi-engines and data-reuse among distributed on-chip buffers, design challenges such as optimal parallelism exploitation and tensor movement orchestration remain to be solved. Therefore, beyond operator-level loop transformation for single engine, dataflow design from a view of the whole DNN computation graph is demanded to gain scalable performance. Although one can evenly split each DNN layer to all the engines and brings the problem back to single engine mapping, this naive approach leads to severe computing resource under-utilization (Fig. 2). Research interests have been devoted on resource partitioning of spatial architectures, such as CNN-Partition [51] and Tangram [19]. However, due to their intrinsic limitations of fixed hardware regions or DNN layer allocation, problems such as workload unbalance and pipeline delay severely affect efficiency, as will be demonstrated in Sec. II.

978-1-6654-2027-3/22/$31.00 ©2022 IEEE
DOI 10.1109/HPCA53966.2022.00042

The basic idea of this paper is the *atomic dataflow*: instead of occupying all the tensor-engines with one or several DNN layers until their completion, we schedule the computation graph in finer granularity (defined as *atoms*) which is specific to DNN topology, layer shapes, and detailed spatial architecture designs, and orchestrate the atom execution in flexible order which could break the initial layer sequence, ensuring PE utilization of each engine and reusing data spatially via dedicated atom-engine mapping. Ideally, this approach can undoubtedly improve the performance of scalable accelerators, as the previous resource allocation schemes are covered in its search space. However, the implementation of atomic dataflow is nontrivial due to the following challenges: (1) Since the efficiency of each engine is greatly influenced by the parameters of its allocated task, the granularity of atoms must be carefully determined, and execution time of parallel engines are expected to be equal to avoid load unbalance. (2) It is nontrivial to sort all the atoms into an optimal execution order since the DAG search space becomes enormous. (3) Breaking layer sequence makes on-chip buffer management more complicated, and brings more inter-engine data reusing options.

To tackle the above challenges, this paper makes the following contributions based on the idea of atomic dataflow:

- We propose an optimizing framework for workload orchestration on scalable accelerators, which supports DNNs with arbitrary network topology (Sec. III).
- We propose simulated annealing based technique to determine optimal atomic granularity given specific DNN workload and spatial architecture (Sec. IV-A).
- We propose dynamic programming based searching algorithm to explore the large graph-level scheduling space and fully exploit the inherent parallelism (Sec. IV-B).
- We propose atom-engine mapping and on-chip buffering schemes to boost spatial data reuse and reduce external memory access. (Sec. IV-C).

II. Background and Motivation

A. Scalable Neural Network Accelerators

Fig. 1(a) illustrates the spatial architecture of a typical DNN accelerating engine, which mainly consists of a two-dimensional processing element (PE) array, global buffer (usually organized as multi-bank SRAM to store feature maps and weights, respectively), engine controller, DMA, etc. The massive multiply-and-accumulate (MAC) operations in DNN are processed by the PE array, and summed in the accumulation unit along each column. Element-wise layers such as ReLu, sigmoid, pooling, and batch-normalization are executed by the vector unit. By broadcasting weights or propagating input feature maps in a systolic manner, this architecture exploits data reuse opportunity in DNN inference along the spatial directions. Operation of the

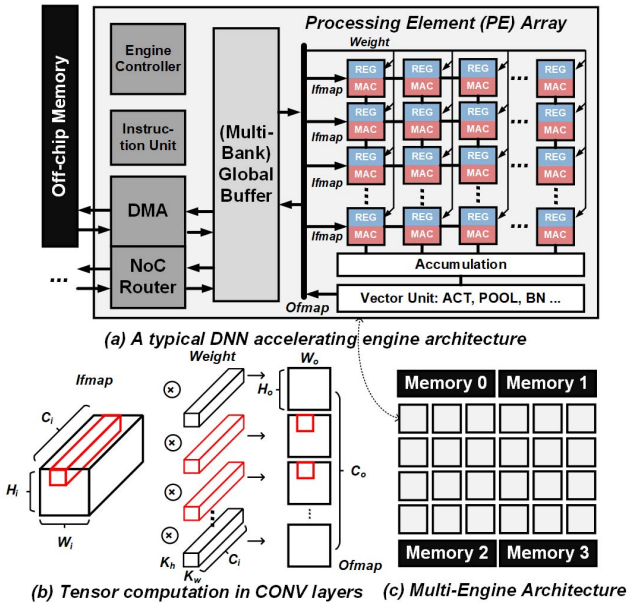


Figure 1. (a) A typical spatial architecture for DNN inference. (b) Definition of CONV layer parameters. (c) Scalable accelerator integrating multi-engines.

engine is ordered by the instructions (or configurations) loaded before execution, which are generated at compile-time since DNN workloads are static. The tensor computation in convolutional (CONV) layers are shown in Fig. 1(b), where the height, width, channel of input and output feature maps (i/ofmaps) are defined as H_i, W_i, C_i, etc. And the sizes of weight kernels are defined as $K_h \times K_w$. Each CONV layer can be partitioned as sub-tasks and executed separately, as exemplified by the red cubes in the figure.

In recent years, with DNN workloads evolving rapidly in compute scale, memory size, and topological complexity, DNN accelerators with scalable performance have attracted strong research interests. A straightforward way of scaling is simply increasing the size of PE array and buffer capacity of single engine. However, such a monolithic computing array has been proved to be performance- and energy- inefficient [19], [50], due to mismatch of fixed array size versus various DNN layer shapes, data propogation cost from edge side PEs to inner PEs, poor wire delay scaling, etc. In contrast, state-of-the-art designs keep the PE array and buffer capacity of an engine at moderate scale, and connect multiple engines via network-on-chip (NoC) to build up multi-engine scalable DNN accelerators [7], [18], [19], [28], [50], [55], as shown in Fig. 1(c).

B. Resource Scheduling Strategies

To clarify our research motivation, we show that simply scaling up computing resources does not bring proportional speedup due to under-utilization. We examine the layer-wise PE utilization rate of a straightforward scheduling strategy: sequentially process DNN layers one-at-a-time and

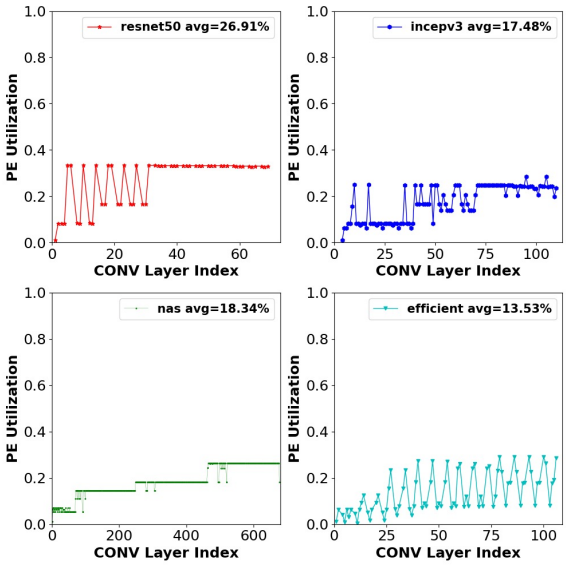


Figure 2. Running DNN layers one-at-a-time (evenly partitioned to all on-chip engines) results in the average PE utilization rates of only 26.91%(ResNet50), 17.48%(Inception_v3), 18.34%(NasNet), and 13.53%(EfficientNet).

evenly partition each layer across multi-engines. Tested on four typical DNN workloads (detailed hardware settings in Sec. V-A), the results are shown in Fig. 2 (the data communication delay is not accounted). As can be observed, this strategy results in severe PE under-utilization of only 13.5–26.9% (layer-averaged).

The reason behind this phenomenon is the mismatch between each engine's microarchitecture and its allocated sub-task: In Layer Sequential **(LS)** scheduling, the computation of each DNN layer is partitioned along certain directions (H_o, W_o, C_o, C_i, etc.) [18] to utilize all engines. However, the PEs in each engine's 2D array cannot be perfectly covered when a sub-task does not reach certain tensor shape threshold (further analyzed in Sec. IV-A). Even in the context of batch processing, it is also infeasible to proportionally scale the throughput of DNN inference by simply using the above LS strategy, because simultaneously processing multiple samples leads to multiplied requirement of off-chip memory bandwidth and on-chip buffer capacity, which may not be satisfied due to the chip's resource constraint (e.g., the global buffer of a Tangram engine is only 32KB [19]). Therefore, beyond the strategy of regarding all engines as a monolithic scheduling unit, more flexible resource allocating strategies are demanded.

From related works [19], [51], [56], [60], we summarize two representative methods: CNN-Partition **(CNN-P)** and Inter-Layer-Pipelining **(IL-Pipe)**. According to CNN-P, the on-chip resources are clustered as convolutional layer processors (CLPs), and multiple DNN layers are allocated to each CLP, e.g. [A,E]→CLP-0. As shown in Fig. 3(a), all the CLPs run in parallel and process DNN layers in one *segment*.

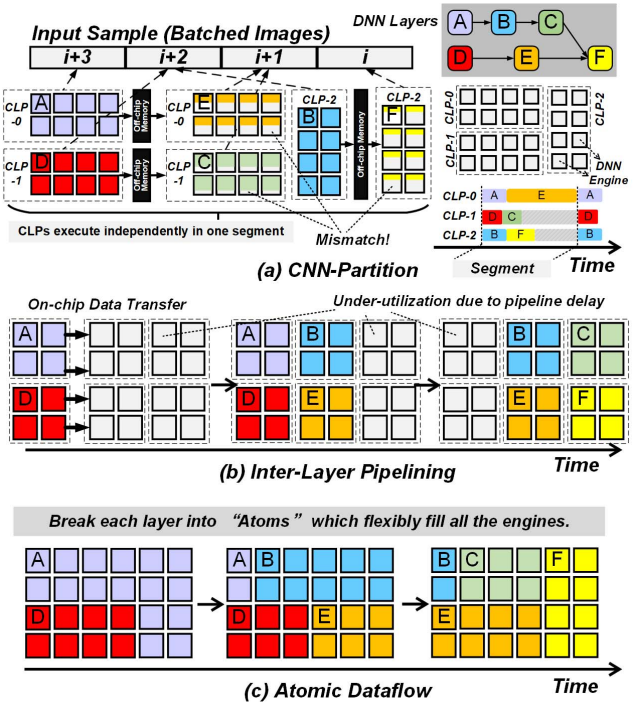


Figure 3. Conceptually comparing multi-engine DNN workload orchestration strategies. A six-layer computation graph is used as an example.

To avoid data dependency inside each segment, processing of batched images is pipelined in layer granularity, e.g. 4 images are in-flight in each segment (Fig. 3). Therefore, each CLP reads inputs and weights from off-chip memory and write the outputs back to it [51]. Since multiple layers with various shapes are bound in one fixed CLP, mismatch issues are likely to happen. Moreover, the segment length is determined by the slowest CLP (even [B,F] finishes before [A,E], CLP-2 must wait in idle because of data dependency). As shown in Fig. 3(b), IL-Pipe eliminates redundant off-chip memory accesses in CNN-P by mapping cascaded DNN layers to adjacent on-chip regions, with all engines partitioned in proportion to the computation cost of each layer. However, as can be observed, this approach suffers from obvious resource under-utilization due to pipeline filling/draining overhead. Even enhanced with fine-grained pipelining, only half of the delay can be alleviated, as analyzed by M. Gao et al. [19].

To overcome the inefficiencies of previous methods, in this paper we propose **atomic dataflow** to orchestrate DNN workloads for multi-engine spatial architectures: It partitions DNN tensor computation into finer-grained scheduling units (atoms) which are expected to perfectly fit engine microarchitecture to ensure high PE utilization, and each layer's atoms can be less or more than the number of total engines. Then, all the atoms are flexibly mapped in the vision of whole atomic computation graph rather than layer-wise sequence, with inherent DNN parallelism being

fully exploited to maximally fill the physical engines, as illustrated in Fig. 3(c). [1]

III. Overview of Atomic Dataflow

To develop the idea of atomic dataflow into a complete workflow which can automatically search and evaluate candidate DNN mapping solutions, we establish an optimization framework which is capable of processing various DNN workloads and hardware settings. In this section, we define the basic concepts of atomic dataflow, introduce overall workflow, and demonstrate the iterative optimizing process.

(1) Concept definition: Each DNN inference workload can be represented as a direct acyclic graph (DAG), with each vertex being an immediate tensor generated by each layer. When its layers are partitioned into atoms, a corresponding **atomic DAG** is also generated as follows,

$$\begin{aligned} G &= (Vertex, Edge) \\ Vertex &= \{Atom_{l,x,(b)} : [(h_s, h_e), (w_s, w_e), (c_s^i, c_e^i), (c_s^o, c_e^o)]\} \\ Edge &= \{e_{l,x,m,y} : Atom_{l,x} \rightarrow Atom_{m,y}\} \end{aligned} \tag{1}$$

where G is a grouping of atomic vertices and edges, each $Vertex$ is the x-th atom partitioned from l-th DNN layer of b-th input sample (omitted when Batch size is one) along height, weight, and channel directions (h_s and h_e represent starting and ending coordinates, respectively), and each $Edge$ represents atom-level data dependency from $Atom_{l,x}$ to $Atom_{m,y}$. All the inferences in a batch are gathered as one unified DAG in our framework, i.e. G is comprised of $\#Batch$ identical sub-DAGs. With atoms being generated, we schedule the graph G in discrete $Rounds$ (as will be illustrated in Fig. 6). Suppose the scalable accelerator consists of N independent engines, in each $Round$, at most N atoms are selected and allocated to separate engines given certain scheduling and atom-engine mapping strategies. These N atoms are synchronized by the last finished one, and be replaced by the new N atoms of next $Round$.

(2) Overall workflow of atomic dataflow is demonstrated in Fig. 4(a). The DNN models imported from mainstream deep learning frameworks are transformed into uniform ONNX format [2]. The necessary information (data dependency, DNN operator types, tensor parameters, etc.) is extracted via our front-end parser to build the initial computation graph for subsequent scheduling, in which networks with arbitrary wiring topology are supported. The core of this framework is comprised of a top-level scheduler, an atomic DAG schedule space exploration module, and a resource allocation module. We build a system evaluation model considering off-chip access, inter-engine data transfer, engine performance, etc., where publicly released tools are integrated (Sec. V-A).

[1] If the problem decrements from scheduling whole DNN graph to mapping single DNN layer, atomic dataflow also decrements to traditional loop tiling.

(3) Iterative optimizing process: Since the 3 stages (atom generation, scheduling, and mapping) are closely linked, we combine them into an iterative searching process. As Fig. 4(b) shows, whenever the DNN workload or HW settings change, new atomic DAG will be generated using techniques in Sec. IV-A. Then, the graph exploring algorithm (Sec. IV-B) iteratively feed the candidate schedule to subsequent module, which finds the optimal atom-engine mapping for every schedule and sends the temporary solution for evaluation. The solution with minimum cost is recorded and will be selected as final solution when the whole atomic DAG has been traversed.

IV. Optimizing Techniques

In this section, we propose the 3 main techniques that optimize the atomic dataflow and boost the efficiency of multi-engine spatial architectures.

A. Atomic Tensor Generation

As analyzed in Sec. II-B, given fixed computing resources and executing mechanism of single engine, the cause of inefficient scheduling is the task-engine mismatch. In this section, we optimize the granularity of atoms with twofold targets: (1) High PE utilization rate of each engine when executing atoms. (2) Since atoms from various layers can possibly run in parallel (scheduled at a same $Round$), they should have close computing delay to avoid load unbalance.

CONV [2] layers are shaped by six loop variables, which have different impacts on the accelerating engine comprised of a 2D PE array (Fig. 1). Typical DNN accelerators spatially unroll two variables along two PE directions, while the other ones are spread at temporal steps. For example, NVDLA [1] unrolls the input channels to PE rows and output channels to columns, which reuses the stationary weights at each PE (defined as KC-Partition in MAESTRO [32]). Another strategy proposed by Shi-diannao [17] spatially unrolls height and width of input feature map (YX-Partition). Therefore, the two spatially unrolled variables have a direct impact on the PE utilization, and thereby should be divisible by the PE array size (denoted as PE_x and PE_y). To this end, we define the size of atomic tensor as $[h_p, w_p, c_p^i, c_p^o] = [c_0, c_1, c_2 \times PE_x, c_3 \times PE_y]$ (cite KC-Partition as an example), where $h_p = h_e - h_s + 1$, $w_p = w_e - w_s + 1, ...$ (equation 1) and c_0 to c_3 are tunable positive integers as coefficients. The other two variables (K_h, K_w), CONV kernel sizes, are not partitioned since their value are usually much smaller than PE_x and PE_y. In such manner, target (1) can be guaranteed.

To fulfill target (2), the problem becomes to find the optimal atomic tensor sizes $[h_p, w_p, c_p^i, c_p^o]$ of each DNN layer to meet an unified execution cycle. Since the tensor

[2] We use CONV layer as an example to present our atomic tensor generating methods. Fully-connected (FC) layers can be regarded as a special form of it by setting $H_o = H_i = W_o = W_i = K_h = K_w = 1$.

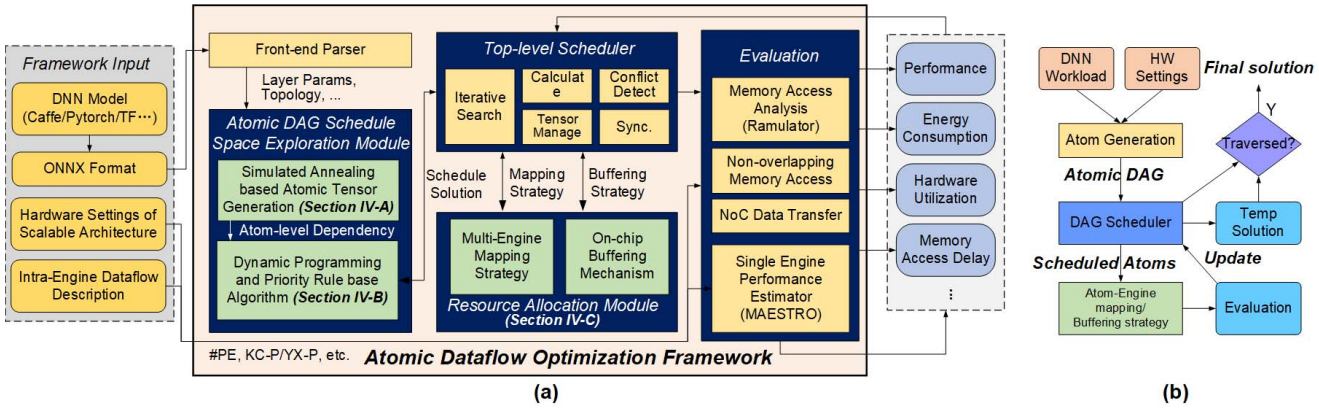


Figure 4. (a) An overview of atomic dataflow optimization framework. (b) Iterative search.

sizes in one workload can be very different, e.g., the fmap sizes vary from 224×224×3 to 7×7×2048 in resnet-152, it is impractical to partition each layer with unified size. Even with the same set of $[h_p, w_p, c_p^i, c_p^o]$, since CONV layers hold various input channels, kernel sizes, strides, etc., the atoms' execution cycles can be very different, which may cause load unbalance. In order to pursuit the optimal coefficients of each layer, we propose a searching algorithm based on simulated annealing (SA) heuristic [30]. We calculate the variance (**Var**) of all atoms' execution cycles, and define it as the system energy. Apparently, the lower **Var** is, the more balanced the parallel atoms will be. We set the system state as the unified execution cycle that each atom is expected to take when executing on single engine, and we scale c_0–c_3 to make execution cycle of each layer's atom get closest to this unified value. In each search step of SA, we consider a neighboring state of current system state, and probabilistically move to this new state according to the energy change (steps that bring lower energy are more likely to be chosen.). The transition continues until the expected energy (**Var** of execution cycle) is met or the iteration reaches upper bound. For better convergence, we gradually decrease the annealing temperature.

The above process is listed in algorithm 1. In line 1–4, we randomly initialize atomic tensor sizes of each layer (represented as four coefficients), and set the iteration upper-bound ite_{max}, maximum movement length Len, convergence condition ϵ, temperature $Temp$ and its decreasing factor λ. In line 5–8, the initial state S is set as the current average execution cycle, and the initial energy E can be obtained correspondingly. The cycle is calculated by feeding $Atom_l$ and l-th DNN layer parameters to a publicly released analyzing tool [3]. After acquiring a neighboring state S^{move} (line 10), we generate new atoms $Atom^{move}$ by adjusting their coefficients to get close with S^{move} (line 11-14). Then, by updating energy and temperature, we calculate the transition probability P and make the movement (line 15–22).

We validate the proposed approach via running each generated atom on single engine and measuring the computing

Input: DNN-layer-param, HW settings, spatial mapping=KC-P/YK-P/...
Output: Atom sizes $[h_p, w_p, c_p^i, c_p^o]$ of each layer
1 **for** *Each layer (l-th)* ***in*** *DNN* **do**
2 **Initialize** $Atom_l = [c_0, c_1, c_2, c_3]_l$
3 **end**
4 **Initialize** ite_{max}, Len, ϵ, $Temp$, λ
5 $S \leftarrow$ ***Mean***(**Cycle**$(Atom_l)$)
6 **//Cycle**: obtain execution cycle via MAESTRO.
7 $E \leftarrow$***Var***(**Cycle**$(Atom_l)$) // *Variance of EXE cycles.*
8 ite = 0
9 **while** $ite++ < ite_{max}$ **do**
10 $S^{move} \leftarrow S + rand(-1, 1) \times Len$
11 **for** *Each layer (l-th) in DNN* **do**
12 $//Atom_l^{move} = [c_0, c_1, c_2, c_3]_l^{move}$
13 $Atom_l^{move} \leftarrow$ $argmin|$**Cycle**$(Atom_l) - S^{move}|$
14 **end**
15 $E^{move} \leftarrow$ ***Var***(**Cycle**$(Atom_l^{move})$)
16 $Temp \leftarrow Temp \times \lambda$
17 $P \leftarrow \exp(\frac{E - E^{move}}{\lambda \times Temp})$
18 **if** *rand(0,1)*$\leq P$ **then**
19 $S \leftarrow S^{move}$
20 $E \leftarrow E^{move}$
21 $Atoms \leftarrow Atoms^{move}$
22 **end**
23 **if** $E \leq \epsilon$ **then**
24 ***exit*** // Terminating iteration.
25 **end**
26 **end**
Result: $Atoms$

Algorithm 1: Simulated Annealing based atomic tensor generation for balanced parallelism.

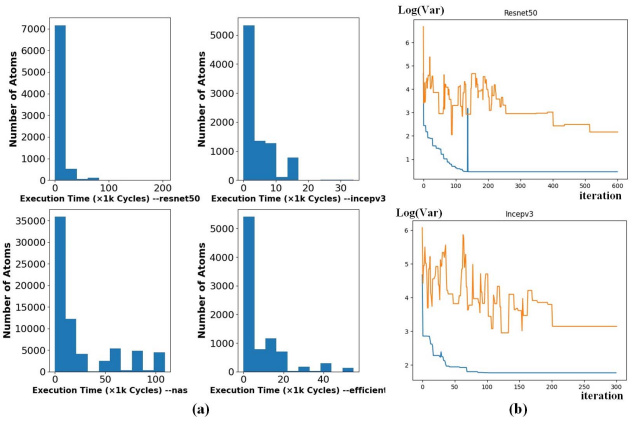


Figure 5. (a) Histograms that show the distribution of atom execution cycles. (b) Convergence trends of SA (blue) and GA (orange).

cycles without accounting data communication delays. As illustrated in Fig. 5(a), tested on each of the four DNN workloads, most of the computing cycles are concentrated to a same region. We also compare SA with genetic algorithm (GA) and show their separate convergence lines in Fig. 5(b), it is clear that SA converges more quickly and stops at lower **Var**. The abrupt rise and fall of GA is due to its mutation operations. The PE utilization rate of the generated atoms will be discussed in Sec. V-B.

B. Atomic DAG Scheduling

In this section, we first study the atom-level data dependency and define four types of parallelism to be exploited. Then, to optimally map the atoms onto parallel engines at each time step, we propose our DAG scheduling algorithm.

As illustrated in Fig. 6(a) [3], we index each layer with a $depth$ value which is the longest path from the source node of DAG to it. As can be observed, all the layers at certain $depth$ can be processed in parallel as long as layers at lower $depth$ are finished. For instance, the three addition layers in (j+1)-th $depth$ can start running after processing layers of j-th and i-th $depth$.

Furthermore, we show the atomic DAG (of the shaded four layers) in Fig. 6(b), in which the output feature maps are partitioned into separate atoms as DAG nodes (omitting channel partitioning for simplicity), the finer-grained atom-level dependencies are also partially marked as DAG edges. It is obvious that an atom can start running immediately when its demanded atoms are ready, rather than waiting for the completion of all previous layer's atoms. This ensures the atom-level parallelism even in cascaded layers (linearly stacked networks such as VGG).

Fig. 6(c) uses a four-engine case to demonstrate a schedule example of Atom-DAG in 6(b), in which the mark X-Y represents the Y-th atom in layer X. From the figure and above analysis, we can draw the conclusion that the parallelism of Atom-DAG scheduling comes from four sources:

[3]We uses PNASNet [39] cell as an example, which is a typical NAS-generated network with irregular topology.

- Intra-layer atoms run in parallel as shown in $Round$ 1,2,4,6.
- Atoms of layers that can be processed directly in parallel (e.g., within same $depth$): in $Round$ 3, since the remaining atom of layer 1 (1-9) cannot fill four engines, scheduler run atoms of layer 2 (at the same $depth$ of layer 1) in parallel to avoid idleness.
- Atoms in dependent layers: in $Round$ 7, although layer 3 is not finished yet, atoms 4-1 and 4-3 are ready for processing because their dependent atoms are already completed, thereby engine 4 can be invoked.
- Batch-level parallelism which simultaneously processes atoms from multiple input samples ($Round$ 8).

For networks with linear structures (VGG), there seems to be no explicit layer parallelism chances. However, atoms of cascaded layers can also be parallelized due to two reasons: (1) The above atom-level parallelism actually incorporates layer fusion [6], which enables atoms in two or more adjacent $depth$s to be executed in parallel. (2) The atomic DAG in our workflow is actually the aggregation of all DNN samples in a batch, i.e. each node is replicated for $\#Batch$ times, which also boosts parallelism beyond initial layer sequence.

To traverse the enormous scheduling space of Atomic DAGs and optimally exploit the above parallelism opportunities, we propose an Atomic DAG scheduling algorithm based on dynamic programming (DP), to determine which atoms to select at each $Round$. We firstly define the optimal substructure of DP: imagine a decision procedure when certain numbers of starting atoms have been scheduled and marked in the DAG, the remaining atoms form a sub-DAG, and apparently, the total execution time can be summed by the already elapsed cycles and the cost of this sub-DAG to be executed in the future. Moreover, the timing cost of the sub-DAG is only determined by the current scheduling state: the computation cost is decided by the remaining atoms themselves, and the communication cost is influenced only by the relevant data presently stored on-chip (ofmaps and weights), which is decided by the buffering strategy (as detailed in Sec. IV-C). Therefore, the optimal substructure is chosen as the un-traversed sub-DAG. Then, in the iterative searching procedure of DP, a candidate set is being updated to indicate the executable atoms at each $Round$, based on the already traversed atoms and atom-level data dependency. Here, the problem becomes which atoms should be chosen from the candidate set, which we discuss later. After the scheduler has chosen atoms to run in parallel from the candidate set, the remaining sub-DAG is re-generated and solved recursively, until the whole Atom-DAG has been traversed.

Assuming the accelerator holds N engines and the number of atoms in candidate set is P, the possible choices at one $Round$ will be C_P^N ($P \geq N$; =1 if $P \leq N$). Exhaustively trying all these decisions will bring impractical searching

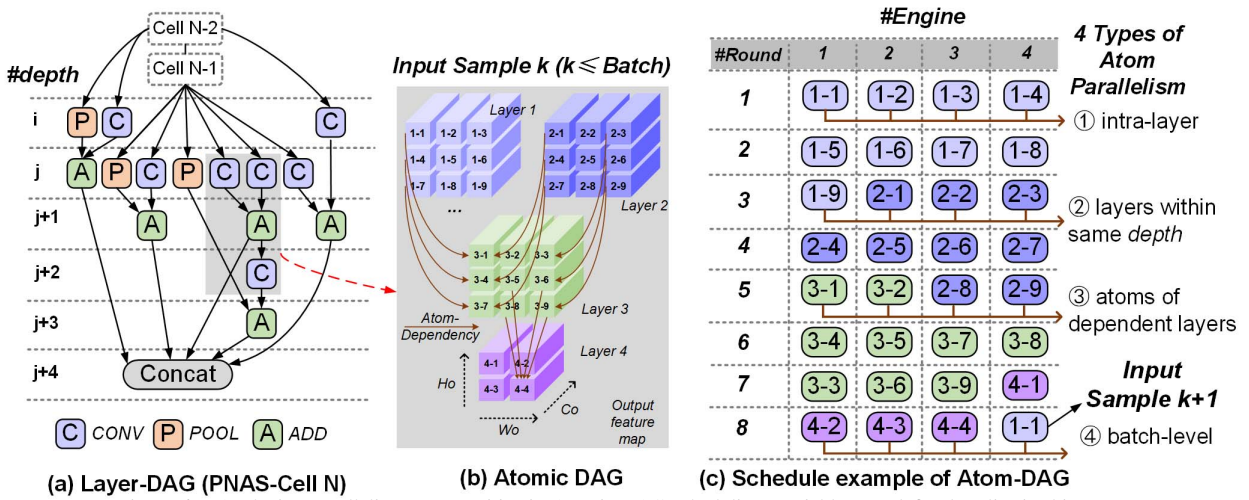


Figure 6. Analyzing parallelism opportunities in Atomic DAG scheduling. Variables are defined earlier in this paper.

time due to combinatorial explosion. Therefore, we prune the combination space by making decision in the order of following ***priority rules*** (in accord with the parallelism types summarized earlier): **(1)** To reuse ifmaps or weights of single layer that have been stored in on-chip buffers, the scheduler prioritizes remaining atoms of traversed layers. **(2)** Since atoms of layers within same depth usually share common inputs, they are prioritized to release the buffer capacity as early as possible. **(3)** Atoms of dependent layer as illustrated in Fig. 6(c) type 3. **(4)** To decrease inference latency, the scheduler does not move to the next DNN sample unless the available atoms of current one cannot occupy all the engines. If available atoms of one priority level is less than N, the scheduler considers next one (1→2→3→4).

The above process is summarized in algorithm 2, which takes the atomic DAG G (inferences in a batch are gathered in one DAG) and hardware settings as input. In line 1–4, $Table$ is initialized to record sub-graph cost, and $CandidateSet$ represents the executable atoms of the untraversed graph G'. Each $Round$ is indexed as t. The recursive function is defined in line 5, which takes an Atomic DAG as input and returns the minimum executing cycle of it. In the iterative search, we firstly update $CandidateSet$ and derive all possible combination of atoms according to our priority rules (line 7-8). Then, we select $Comb_i$ that has the minimum cost (line 10-15) and update the result $Schedule[t]$. The recursion happens when the cost of each $Comb_i$ is calculated (line 11). The final scheduling solution is returned when G is traversed (G' becomes $\emptyset$).

Input: Atom-wise DNN topology: G, # Engines: N.
Output: $Schedule[t]$ // Indicate atoms chosen at each $Round$ t.
1 $Table \leftarrow \emptyset$ // Record minCost of traversed sub-graphs.
2 $CandidateSet \leftarrow \emptyset$ // Executable atoms at each $Round$.
3 $G' \leftarrow G$ // Un-visited Atom-DAG.
4 t←0 // Indicate t-th $Round$ scheduling.
5 **Recursive Function:** $minCost$=**DP**(G) // **DP**($\emptyset$)=0
6 **while** G' *is not empty* **do**
7 **Update** $CandidateSet$ **with** G'.
8 **Update** $Options = \{Comb_i\}$ **with** priority rules. // Each element in $Options$ is a combination of at most N atoms in $CandidateSet$.
9 $minCost \leftarrow \infty$, $optComb \leftarrow \emptyset$
10 **for** ***each*** $Comb_i$ ***in*** $Options$ **do**
11 $cost$ = $Cycle(Comb_i)$ + **DP**($G' \setminus Comb_i$)
12 **if** $cost \leq minCost$ **then**
13 $minCost \leftarrow cost$
14 $optComb \leftarrow Comb_i$
15 **end**
16 **end**
17 **Update** $Table$
18 $G' \leftarrow G' \setminus optComb$
19 $Schedule[t] \leftarrow optComb$
20 $t \leftarrow t$+1
21 **end**
Result: $Schedule$

Algorithm 2: Scheduling Atomic DAGs based on Dynamic Programming.

C. Atom-Engine Mapping and Buffering Strategy

In scalable DNN accelerators, interconnect between multi-engines such as 2D-mesh, H-tree, and Torus [8], [19], [52] enable rich on-chip data reuse opportunities, which distinguishes them from SIMD processors which exchange data via a large shared memory. Our NoC model is based on the 2D-mesh static network (STN) of TILE64 architecture

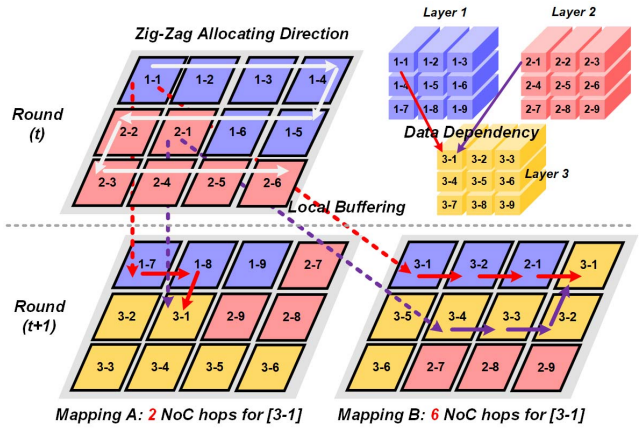


Figure 7. Optimized mapping reduces inter-engine data transfer cost (exemplified with 2D-mesh topology and zig-zag task allocation).

[8], which has single-cycle hop latency between adjacent engines. The switch at each engine is implemented as a full crossbar to connect cardinal directions. A dimension-ordered routing policy is adopted, where data always travel in the X-direction first, then the Y-direction. To avoid deadlock, credit-based flow control is adopted.

Atom-Engine Mapping: Given well ordered atoms that are executed in certain *Rounds*, the total NoC hops are first determined by the placement of atoms. In atomic dataflow, the intermediate tensors (ofmaps and weights) are generated or stored in each engine's distributed buffer, which naturally provides the opportunity of spatial data reuse. However, since each atom can be allocated to any available engine, the NoC hops between the provider and receiver engines varies with different atom-engine mapping strategies.

We show such influence in Fig. 7: with the commonly used zig-zag approach being adopted, atoms scheduled at each $Round$ are allocated onto the 2D array along one logical direction (shown as white arrows), the atoms of same layer are mapped to the adjacent engines. In this example, the execution of $Atom_{3-1}$ depends on $Atom_{1-1}$ and $Atom_{2-1}$, which are processed in $Round(t)$ and stored in the 1st and 7-th engines (counted along zig-zag direction), respectively. We consider two atom-engine mapping solutions. In solution A which maps the atoms in the order of increased layer index (1→2→3), $Atom_{1-1}$ is transferred from the 1st engine to 7-th one where $Atom_{3-1}$ is located, and $Atom_{2-1}$ is reused locally, which consumes 2 NoC hops in total. While in solution B which has the allocation order of (1→3→2), $Atom_{3-1}$ is located at the 4-th engine, and 4 more NoC hops are needed. We formulate the quantitative impact of mapping strategies to inter-engine data transfer as

$$TransferCost(P) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} D(i,j) \times Size(Atom_{l-x}) \quad (2)$$

, where N is number of engines, $D(i,j)$ represents the shortest NoC hops from the i-th to the j-th engine (with 2D mesh topology in this paper), $TensorSize$ equals zero when there is no data transfer between them, and P is a permutation of involved layers[4] at the current $Round$. The permutation with minimum $TransferCost$ is selected as our final mapping strategy.

Buffering Strategy: Another factor that determines on-chip data reuse is the buffering strategy. Considering long-term data dependency in complex DNN structures, the newly produced atoms at each engine may not be immediately reused in the next $Round$, and requires to be stored in buffer for a long time, such as the ADD layer in Fig. 6(a)-$depth$-j. With the gradual accumulation of such data, a new atom generated on an engine may cause buffer overflow. At this moment, the scheduler must decide which atoms to be kept on-chip and which should be written to external memory. Simply writing all atoms with long-term dependency to off-chip memory is obviously not the optimal strategy. Due to the static nature of DNN workloads, this problem can be optimized during compile-time.

Our buffering strategy is based on the estimation of invalid buffer occupation, which is calculated as the product of (1) data size of an atom and (2) the minimum time an atom must wait until it can be reused. In algorithm 3 line 1–2, the occupation and candidate atom are initialized and atoms in $Buffer[n]$ are traversed. Firstly, the earliest reuse $Round$ of currently stored $Atom_{l-x}$ is found in line 3–7 as t_{next}, atoms that are no longer needed will be released from $Buffer[n]$ without write-back (line 8–12). Then, the algorithm estimates the invalid occupation by multiplying time difference and $TensorSize$, and update corresponding variables (line 13–17).

V. Evaluation

Sec. V-A introduces our adopted DNN workloads, performance profiling tools, system modeling methods, and baselines for comparison in the experiments. We demonstrate the effectiveness of atomic dataflow in Sec. V-B in terms of both performance and energy efficiency, and then evaluate the architectural design parameters in Sec. V-C.

A. Methodology

Workloads: To comprehensively evaluate the optimizing strategies, we adopt 8 state-of-the-art DNN models with various computational scales and typical structural characteristics, as summarized in table I.

Hardware Modeling: We set a scalable DNN accelerator with 8×8 engines, each containing 16×16 PEs (totally 16384 PEs by default, the same with a 128×128 monolithic array), the global buffer (SRAM) capacity on each engine is 128KB with the port width being 64b, and the frequency

[4]Assuming M layers are involved, there are $M!$ choices for P.

Input: G=($Vertex, Edge$), $Schedule$, t_0, atoms stored in n-th engine: $Buffer[n]$.
Output: An atom in $Buffer[n]$, which is to be written to off-chip memory.

$occupation \leftarrow \infty$, $Atom\text{-}wb \leftarrow Atom_{0-0}$ // Initialize.
for ***each*** $Atom_{l-x}$ ***in*** $Buffer[n]$ **do**
  $t_{next} \leftarrow t_{max}$ // Index upper bound of $Schedule$.
  for ***each*** $e_{l-x,m-y}$ ***in*** $Edge$ **do**
    $t \leftarrow$**Index**($Atom_{m-y}$ in $Schedule$)
    $t_{next} \leftarrow \min(t, t_{next})$
  end
  if $t_{next} \geq t_{max}$ **then**
    // Realease buffer.
    $Buffer[n] \leftarrow Buffer[n] \setminus Atom_{l-x}$
    Continue
  end
  $occupation_{l-x} \leftarrow (t_{next} - t_0) \times TensorSize(Atom_{l-x})$
  if $occupation_{l-x} \geq occupation$ **then**
    $occupation \leftarrow occupation_{l-x}$
    $Atom\text{-}wb \leftarrow Atom_{l-x}$
  end
end
Result: $Atom\text{-}wb$

Algorithm 3: Buffering strategy for atomic dataflow.

Table I
DNN WORKLOAD CHARACTERIZATION

DNN Model	# Layers	# Params	Characteristics
VGG-19	23	137M	layer cascaded
ResNet-50	73	26M	residual bypass
ResNet-152	516	60M	residual bypass
ResNet-1001	1329	850M	residual bypass
Inception-v3	313	27M	branching cells
NasNet	1232	89M	NAS-generated
PNASNet	914	86M	NAS-generated
EfficientNet	288	2M	NAS-generated

is 500MHz. We select a 4-layer HBM stack as the off-chip memory, which provides a total capacity of 4GB and peak bandwidth of 128GB/s [53]. We adopt MAESTRO [3] to obtain the execution cycle and power consumption of each DNN accelerating engine, which is an open-source model commonly used in schedule space exploration [26], [27]. We use Ramulator [29] to obtain the cycle cost of HBM reads/writes by feeding the accessing traces. Based on these information, we build up an event-driven simulator to evaluate total execution cost of scalable DNN accelerators, and calculate the resource utilization rate. We get the SRAM power information from the datasheet-tt0p9v25c of TSMC 28nm library. For example, the read power of 128KB SRAM is 10.96mW (0.9V, 24.3uA/MHz). The NoC energy is set as 0.61pJ/bit per hop [19], and according to Cacti-3dd [9], the access energy of the adopted HBM stack is 7pJ/bit.

Baseline: Layer-Sequential **(LS)**, CNN-Partition **(CNN-P)** [51], and Inter-layer Pipelining **(IL-Pipe)** [19] are chosen to compare with the Atomic Dataflow **(AD)** proposed in this paper. In addition to the definitions of these approaches in Sec. II-B, we enhance the naive LS method by simultaneously mapping multiple input samples to improve utilization in the context of batch processing, and adopts fine-grained pipelining techniques to reduce filling/draining delay of IL-Pipe [5].

B. Performance and Energy Comparison

We adopt KC-Partition and YX-Partition as two typical DNN layer partitioning strategies on single engine [32], which correspond to NVDLA and ShiDianNao dataflows, respectively [1], [17]. The ideal performance which assumes perfect hardware utilization and zero memory delay is also considered. Since DNN inference workloads are deterministic, we generate the scheduling and mapping solutions during compile-time. Tested on Intel-Xeon-CPU-E5-2620-v3(2.40GHz), AD's searching overheads are 66.5s(resnet50), 102.7s(resnet152), 406.9s(inception-v3), 1044.6s(resnet1001), etc.

Firstly, we evaluate the inference latency (BatchSize=1) of each strategy as shown in Fig. 8. In this case, CNN-P cannot pipeline layers among CLPs, and its mapping strategy is the same with LS (omitted in figure). Evidently, AD achieves significant inference latency speedup over CNN-P and IL-Pipe by 1.45–2.30× and 1.42–3.78×, respectively (with KC-Partition strategy, the situation is similar on YX-Partition case). The reason is the task-engine mismatch of LS and the filling/draining delay of IL-Pipe, which make them not suitable for latency-critical scenarios, as discussed in Sec. II-B. Since atomic dataflow supports four types of parallelism as illustrated in Fig. 6, although batch-level parallelism cannot be exploited, the parallel layers in DNN workloads (except for VGG) and atoms of dependent layers can still ensure the high utilization of computing resources. For VGG (without explicit parallel layers), the improvement is mainly due to (1) implicit layer fusion optimization of DP-based atomic DAG scheduling and (2) PE utilization-aware atom selection rather than evenly partition each layer to all engines.

Then, in the throughput targeted experiments illustrated in Fig. 9, CNN-P has shown its advantage on throughput optimization and exceeds LS in all cases. However, due to its load unbalance and mismatch disadvantages, its throughput is still lower than that of AD, which is 1.12–1.38× and 1.08–1.42× higher than it on KC-P and YX-P, respectively. Since the design target of IL-Pipe is mainly to reduce off-chip memory access and improve the accelerator's energy efficiency, spatial regions on it must wait for prior regions to get ready, which harms the utilization (It is even slower than LS when processing NASNet and PNASNet of KC-Partition

[5]Referred to as Alternate Layer Loop Ordering (ALLO) in Tangram [19].

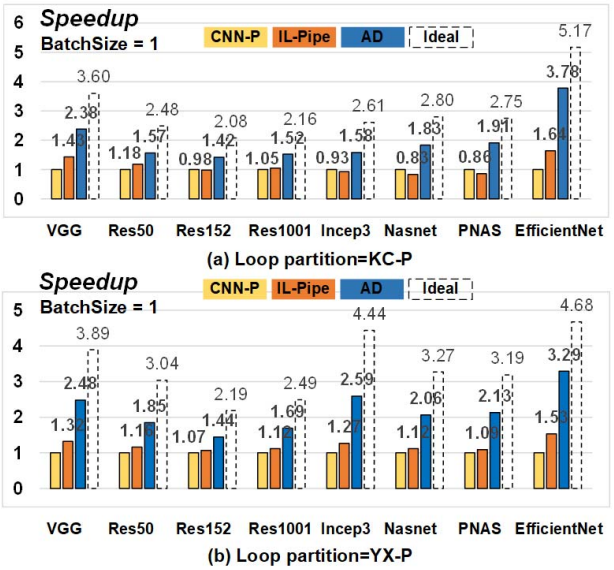


Figure 8. Evaluation of DNN inference latency.

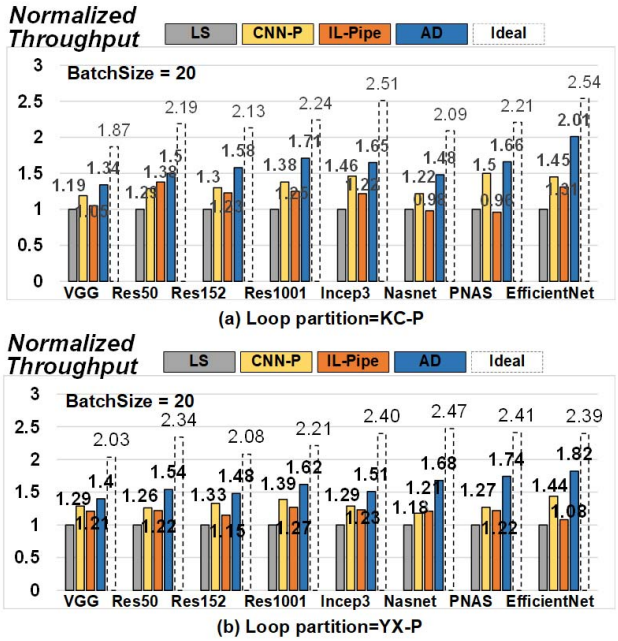


Figure 9. Evaluation of DNN inference throughput with batch size being 20.

case.). In contrast, as the tensor computation of each DNN layer is carefully partitioned and flexibly scheduled with the proposed optimizing techniques in Sec. IV, therefore on-chip computing resources can be well utilized.

We illustrate per-stage performance improvements in Fig. 10. The DP based DAG scheduling algorithm accounts for the most significant improvement of 1.17-1.42×. The SA based atom generation and on-chip data reuse mechanisms contributes 1.06-1.21× and 1.07-1.17×, respectively.

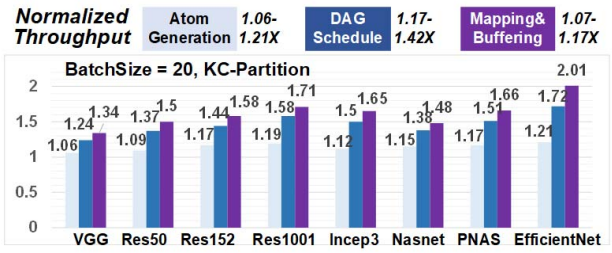


Figure 10. Per-stage performance improvements.

To directly reflect whether sub-tasks and their corresponding engines are well matched, we summarize the computing cycles and list the overall PE utilization rates without accounting communication delay in Tab. II (the numbers in this table do not correspond to the performance ratio in the above figures). It can be observed that CNN-P achieves good matching of layers and CLPs. However, since in CNN-P each ifmap and ofmap inevitably introduces off-chip memory access, the expensive DRAM reading and writing dismisses its advantage, which cannot be completely overlapped by double buffering. The high utilization of AD in this table is enabled by our simulated annealing based atom generating algorithm, which guarantees the efficiency via the analysis of DNN accelerator microarchitectures. We also list the NoC overhead in the total time cost (the part that blocks the computation of engines), benefited from our atom-engine mapping method, it ranges only from 9.4% to 17.6%. To show how many data are reused on-chip rather than stored externally, we list such ratios (54.1-90.8%) at the last row of Tab. II, which indicate significant reduction of external memory access.

In Fig. 11 we evaluates energy consumption of 4 strategies. As expected, IL-Pipe and AD are the most energy-efficient approaches. Since in atomic dataflow, write-backs are demanded only when buffer overflow occurs, it consume slightly more energy than IL-Pipe on the first 3 workloads due to larger amount of off-chip access and on-chip inter-engine data transfer. However, due to the buffering algorithm 3 and the mapping strategy to obtain minimum on-chip data transfer (NoC hops), AD still achieves lower energy consumption than IL-Pipe on the other four workloads. The shorter execution time (less static power consumption) also contributes to these results.

C. Architectural Design Space Exploration

We utilize our framework to facilitate the design space exploration of scalable DNN accelerators. Given a fixed total number of PEs (16384) and on-chip buffer capacity (8MB), in Fig. 12 we show the execution time of each DNN workload which varies with the increasing number of engines (decreasing PE number and buffer size of each engine). It can be observed that the curves are all in the ”U” shape, and for each DNN model, there is a sweet-point of the total engines, which correspondingly determines the

Table II
(1) PE UTILIZATION AVERAGED AMONG DNN LAYERS W/O MEMORY ACCESS DELAY (BATCHSIZE=20). (2) NOC OVERHEAD AND ON-CHIP REUSE.

Method	VGG-19	ResNet-50	ResNet-152	Inception-v3	NasNet	PNasNet	EfficientNet	ResNet-1001
LS	64.3%	51.1%	60.3%	49.0%	63.7%	69.2%	59.2%	56.4%
CNN-P	74.2%	59.5%	70.3%	57.4%	76.7%	79.8%	72.8%	72.8%
IL-Pipe	57.1%	47.4%	64.6%	45.7%	58.7%	61.8%	53.7%	67.7%
AD	86.0%	82.2%	88.9%	78.8%	85.0%	90.2%	95.0%	91.5%
NoC Overhead (AD)	15.6%	13.2%	15.4%	10.0%	15.9%	17.6%	9.4%	10.5%
On-chip Data Reuse Ratio (AD)	60.2%	82.0%	85.7%	90.8%	54.1%	61.2%	87.3%	79.6%

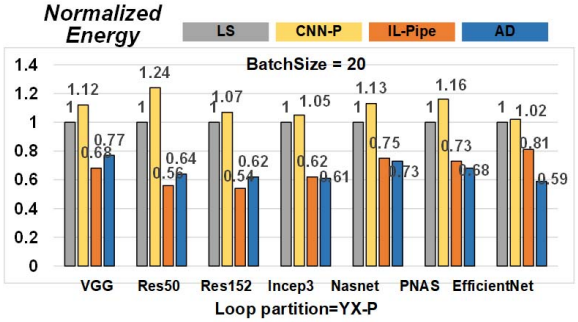


Figure 11. Evaluation of DNN inference energy consumption with batch size being 20.

scale of PE and buffer. For example, when PEs and SRAM are partitioned into 4×4 engines, VGG-19, ResNet-152 and NasNet reach their minimum execution time. This is a comprehensive phenomenon: while monolithic PE array suffers from under-utilization (Sec. II), PE array that is too small will reduce the data reuse opportunities along row/columns and thereby induce increased amount of data movement (as shown by Fig. 1). Although we conduct the experiments under two batch sizes, it seems that the doubled batch size does not change the trend of performance variation. Another question is: if we provide larger on-chip buffers (which means increased chip area), how will the performance be boosted? As shown in Fig. 13, the results do benefits from increased buffer size. But apparently, such trends slow down when exceeding 128KB. It can be inferred that this growth is not unlimited. Due to our data transferring and reusing techniques for the distributed buffers, the performance is not greatly influenced with small memory, since they can be efficiently utilized.

D. Evaluation on FPGA Prototype

To demonstrate the efficacy of atomic dataflow on real systems, we build an accelerator prototype with 2x2 engines on Synopsys HAPS platform. Each single engine runs at 600MHz with power consumption being 370mW, supports up to 32x32 INT8 MACs in each cycle, and has been verified after fabrication (Fig. 14). Performance of VGG on four engines are measured as 49.2fps(**LS**), 57.9fps(**Rammer**), and 64.3fps(**AD**). For ResNet50, the numbers are 156.2fps(**LS**), 194.4fps(**Rammer**), and 223.9fps(**AD**). The AD's improvements over baseline are close to the results evaluated by our simulation-based methodology.

VI. RELATED WORKS AND DISCUSSION

A. DNN accelerators

The prevalence of deep neural networks has attracted great interests of domain-specific DNN accelerators [5], [16], [17], [20], [21], [25], [31], [36], [38], [44], [46], [58]. An important aspect to classify them is the customized data reusing mechanism (aka *dataflow*): Eyeriss [12], [13] analyzes the trade-offs in intra-layer data-reuse and proposes row-stationary dataflow, Fused-CNN [6] presents a fused-layer scheduling approach to reduce energy-consuming off-chip memory accesses, FlexFlow [41] and MAERI [34] support flexible dataflow strategies via micro-architectural reconfiguration to adapt themselves to various DNN layer shapes. Scalable DNN accelerators are proposed to support high performance deep learning applications [55]. Scale-sim studies the scalability of systolic-array based DNN accelerators and gives sweet points for hardware configurations [49]. Neurocube [28] and TETRIS [18] integrate logic dies with vertical memory chips to overcome memory bottleneck, they partition each DNN layer across logic dies to exploit intra-layer parallelism. Tangram [19] develops a coarse-grained dataflow to enable inter-layer pipelining. HDA supports to deploy DNN onto heterogeneous sub-accelerators [33]. In this paper, we further improve the utilization rate of scalable DNN accelerators by proposing and optimizing atomic dataflow.

Discussion: While more powerful arrays that can spatially map more than 2 loop parameters and flexibly switch between fine-grained dataflows have been proposed, they can also benefit from atomic dataflow. The key adaptation is to merely change the atoms' coefficients in searching procedure (Sec. IV-A) to their forms, e.g., for dataflow additionally support width splitting beyond KC-Partition: $[h_p, w_p, c_p^i, c_p^o] = [c_0, c_1 \times PE_z, c_2 \times PE_x, c_3 \times PE_y]$.

B. Scheduling and mapping space exploration

After DNN accelerators are designed and fabricated, the scheduling and mapping strategies that control the execution of DNN models significantly impacts the hardware latency, throughput, and energy efficiency. To this end, extensive prior works have been devoted to search for optimal strategies. Since each scheduling solution of single DNN layer can be represented as nested loops transformed from the original ones, a scheduling space can be formed by considering

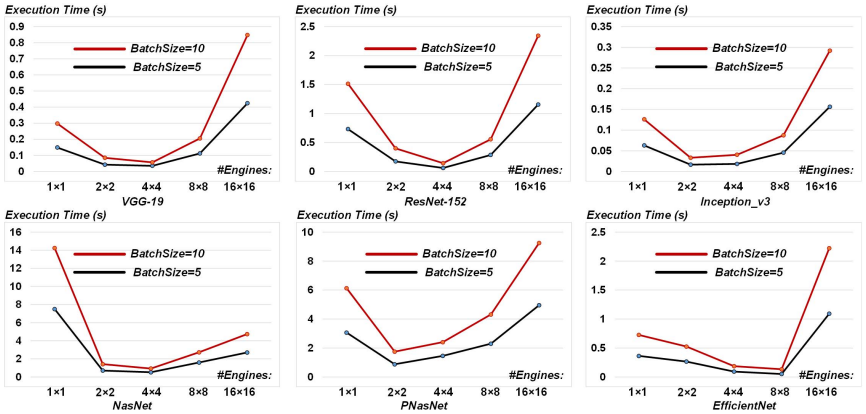


Figure 12. Scaling the number of engines while maintaining the total number of PEs and buffer capacity.

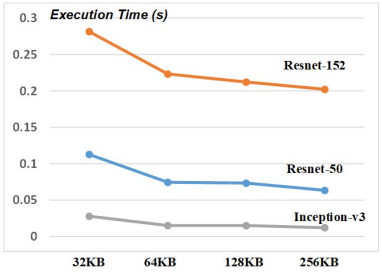
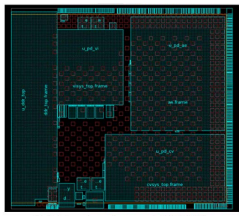


Figure 13. Scaling buffer size on each engine.

CLB LUTs	CLB Registers	CARRY8
385893	136709	5305
CLB	LUT as Logic	DSPs
63082	380837	98
LUT as Memory	LUT Flip Flop Paris	
5056	101615	
Block RAM Tile	F7/F8 Muxes	
269.5	216	

(a) FPGA synthesis results of single engine, which we use to build a 2x2 accelerator.



(b) Engine layout (ASIC).

Figure 14. Information of single engine in the prototype system.

tiling, re-ordering, unrolling, etc. [35], [37], [43], [57], [59]. Schedulers for co-locating or time-multiplexed multi-DNNs have also been proposed [7], [14], [45]. To explore this large space, heuristic or machine learning based algorithms are proposed, such as the boosted tree search and simulated annealing-based AutoTVM [10], [11], the genetic algorithm-based TensorComprehensions [54] and GAMMA [27], polyhedral model based PolySA [15], reinforcement learning-based FlexTensor [63] and ReLeASE [4], and bayesian optimization [47]. To exploit inter-layer data locality in DNN inference and eliminate unnecessary off-chip memory access [6], node-clustering and dynamic programming-based algorithm is proposed for irregular network structures [62]. TASO [24] uses substitution methods to optimize DNN computational graphs. Polyhedral models are also adopted to determine tiling and fusing strategies [61]. To efficiently allocate on-chip resources, layer-wise parallelism has been studied by CNN-Partition [51], TGPA [56], DNNBuilder, [60], Tangram [19], etc. Although these approaches have alleviated the mismatch problem, load unbalance (due to fixed mapping regions) and resource under-utilization (due to pipeline delay) are not well addressed yet. For multi-chip mapping, pioneering framework like NN-Baton has also been proposed, but it focuses on layer-wise loop-parameter tuning rather than graph level scheduling discussed in this work. Rammer boosts GPU utilization by co-locating multiple rTasks, but it does not discuss how the rTasks are generated, nor does it consider spatial data reuse, inter-array communication, engine resources partitioning, and layer fusion [42]. In comparison, the atomic dataflow optimizing framework proposed by this paper schedules DNNs in PE utilization-oriented granularity, enables flexible mapping and exploits data locality, which further boost the performance and efficiency of scalable multi-engine DNN accelerators.

VII. Conclusion

This paper proposes a novel DNN workload orchestration methodology for scalable accelerators with multi-engine spatial architectures. With the proposed SA based atomic tensor generation, DP based atomic DAG scheduling, atom-engine mapping strategy, and buffering mechanism, the DNN inference performance, hardware utilization rate, and energy efficiency have been significantly improved.

Acknowledgment

This work was supported by the NSFC (U19B2041 and 62125403), National Key R&D Project (2018YFB2202600), the Beijing S&T Project (Z191100007519016) and Beijing Innovation Center for Future Chips.

References

[1] "Nvdla deep learning accelerator," http://nvdla.org, 2017.

[2] "Open neural network exchange," http://onnx.ai, 2018.

[3] "Maestro tool," http://maestro.ece.gatech.edu/, 2020.

[4] B. H. Ahn, P. Pilligundla, and H. Esmaeilzadeh, "Reinforcement learning and adaptive sampling for optimized DNN compilation," *arXiv preprint arXiv:1905.12799*, 2019.

[5] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 382–394.

[6] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 22.

[7] E. Baek, D. Kwon, and J. Kim, "A multi-neural network acceleration architecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 940–953.

[8] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "Tile64 - processor: A 64-core soc with mesh interconnect," in *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, 2008, pp. 88–598.

[9] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, pp. 33–38.

[10] 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[11] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," *arXiv preprint arXiv:1805.08166*, 2018.

[12] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks," *arXiv preprint arXiv:1807.07928*, 2018.

[13] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[14] Y. Choi and M. Rhu, "Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 220–233.

[15] J. Cong and J. Wang, "Polysa: Polyhedral-based systolic array auto-compilation," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[16] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan, "CirCNN: accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 395–408.

[17] 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 *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 92–104.

[18] 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 Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 751–764.

[19] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, "Tangram: Optimized coarse-grained dataflow for scalable nn accelerators," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 807–820.

[20] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[21] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "UCNN: Exploiting computational reuse in deep neural networks via weight repetition," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 674–687.

[22] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7132–7141.

[23] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *arXiv preprint arXiv:1811.06965*, 2018.

[24] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 47–62.

[25] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, "A domain-specific architecture for deep neural networks," *Communications of the ACM*, vol. 61, no. 9, pp. 50–59, 2018.

[26] S.-C. Kao, G. Jeong, and T. Krishna, "Confuciux: Autonomous hardware resource assignment for DNN accelerators using reinforcement learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 622–636.

[27] S.-C. Kao and T. Krishna, "Gamma: automating the hw mapping of DNN models on accelerators via genetic algorithm," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.

[28] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 380–392, 2016.

[29] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[30] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[31] H. Kung, B. McDanel, and S. Q. Zhang, "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*, 2019, pp. 821–834.

[32] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 754–768.

[33] H. Kwon, L. Lai, M. Pellauer, T. Krishna, Y.-H. Chen, and V. Chandra, "Heterogeneous dataflow accelerators for multi-dnn workloads," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 71–83.

[34] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 461–475, 2018.

[35] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 242–251.

[36] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H. Yoo, "Unpu: A 50.6tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, 2018, pp. 218–220.

[37] R. Li, Y. Xu, A. Sukumaran-Rajam, A. Rountev, and P. Sadayappan, "Analytical characterization and design space exploration for optimization of CNNs," *arXiv preprint arXiv:2101.09808*, 2021.

[38] H. Liao, J. Tu, J. Xia, and X. Zhou, "Davinci: A scalable architecture for neural network computing," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, 2019, pp. 1–44.

[39] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *The European Conference on Computer Vision (ECCV)*, September 2018.

[40] L. Liu, Z. Qu, L. Deng, F. Tu, S. Li, X. Hu, Z. Gu, Y. Ding, and Y. Xie, "Duet: Boosting deep neural network efficiency on dual-module architecture," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 738–750.

[41] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 553–564.

[42] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou, "Rammer: Enabling holistic deep learning compiler optimizations with rtasks," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 881–897.

[43] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 45–54.

[44] B. Moons and M. Verhelst, "An energy-efficient precision-scalable convnet processor in 40-nm cmos," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, pp. 903–914, 2017.

[45] Y. H. Oh, S. Kim, Y. Jin, S. Son, J. Bae, J. Lee, Y. Park, D. U. Kim, T. J. Ham, and J. W. Lee, "Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 584–597.

[46] 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," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[47] B. Reagen, J. M. Hernández-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G.-Y. Wei, and D. Brooks, "A case for efficient accelerator design space exploration via bayesian optimization," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2017, pp. 1–6.

[48] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the aaai conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 4780–4789.

[49] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "A systematic methodology for characterizing scalability of dnn accelerators using scale-sim," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 58–68.

[50] 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 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 14–27.

[51] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 535–547.

[52] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Hypar: Towards hybrid parallelism for deep learning accelerator array," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 56–68.

[53] M. Ujaldón, "Hpc accelerators with 3d memory," in *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*. IEEE, 2016, pp. 320–328.

[54] 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," *arXiv preprint arXiv:1802.04730*, 2018.

[55] 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 44th Annual International Symposium on Computer Architecture*, 2017, pp. 13–26.

[56] X. Wei, Y. Liang, X. Li, C. H. Yu, P. Zhang, and J. Cong, "Tgpa: tile-grained pipeline architecture for low latency CNN inference," in *Proceedings of the International Conference on Computer-Aided Design*, 2018, pp. 1–8.

[57] X. Yang, J. Pu, B. B. Rister, N. Bhagdikar, S. Richardson, S. Kvatinsky, J. Ragan-Kelley, A. Pedram, and M. Horowitz, "A systematic approach to blocking convolutional neural networks," *arXiv preprint arXiv:1606.04209*, 2016.

[58] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "Ganax: A unified mimd-simd acceleration for generative adversarial networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 650–661.

[59] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "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, 2015, pp. 161–170.

[60] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for fpgas," in *Proceedings of the International Conference on Computer-Aided Design*. ACM, 2018, p. 56.

[61] J. Zhao and P. Di, "Optimizing the memory hierarchy by compositing automatic transformations on computations and data," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 427–441.

[62] S. Zheng, X. Zhang, D. Ou, S. Tang, L. Liu, S. Wei, and S. Yin, "Efficient scheduling of irregular network structures on CNN accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3408–3419, 2020.

[63] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 859–873.

[64] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.