KAPLA: Scalable NN Accelerator Dataflow Design Space Structuring and Fast Exploring

Zhiyao Li
lizhiyao19@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Mingyu Gao
gaomy@tsinghua.edu.cn
Tsinghua University
Beijing, China
Shanghai Artificial Intelligence Lab
Shanghai, China
Shanghai Qi Zhi Institute
Shanghai, China

ABSTRACT

Dataflow scheduling is of vital importance to neural network (NN) accelerators. Recent scalable NN accelerators support a rich set of advanced dataflow techniques. The problems of comprehensively representing and quickly finding optimized dataflow schemes thus become significantly more complicated and challenging. In this work, we first propose comprehensive and pragmatic dataflow representations for temporal and spatial scheduling on scalable multi-node NN architectures. An informal hierarchical taxonomy highlights the tight coupling across different levels of the dataflow space as the major difficulty for fast design exploration. A set of formal tensor-centric directives accurately express various inter-layer and intra-layer schemes, and allow for quickly determining their validity and efficiency. We then build a generic, optimized, and fast dataflow solver, KAPLA. It makes use of the pragmatic directives to explore the design space with effective validity check and efficiency estimation. KAPLA decouples the upper inter-layer level for fast pruning, and solves the lower intra-layer schemes with a novel bottom-up cost descending method. KAPLA achieves within only 2.2% and 7.7% energy overheads on the result dataflow for training and inference, respectively, compared to the exhaustively searched optimal schemes. It also outperforms random and machine-learning-based approaches, with more optimized results and orders of magnitude faster search speed.

CCS CONCEPTS

• **Hardware** → **Hardware accelerators**; • **Computing methodologies** → *Artificial intelligence*; • **Computer systems organization** → *Special purpose systems.*

KEYWORDS

hardware acceleration, deep learning, dataflow

ACM Reference Format:
Zhiyao Li and Mingyu Gao. 2025. KAPLA: Scalable NN Accelerator Dataflow Design Space Structuring and Fast Exploring. In *30th Asia and South Pacific Design Automation Conference (ASPDAC '25), January 20–23, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3658617.3697549

1 INTRODUCTION

As Dennard scaling ends, modern computing systems start to adopt domain-specific accelerators alongside general-purpose processors [6, 12, 29]. With neural networks (NNs) evolving to feature larger and more complex model structures [8], we would now need scalable accelerator architectures with greater on-chip compute and storage resources for improved performance and efficiency [7, 29]. Fully utilizing these abundant resources has thus motivated a rich set of new dataflow techniques in recent designs [6, 7, 12, 26, 39].

However, achieving high resource utilization with optimized dataflow scheduling on scalable NN accelerators is challenging. There has not been a comprehensive framework to systematically capture the extensive design space of NN dataflow schemes, nor an effective methodology to quickly identify optimized schedules for specific workloads and hardware. Previous solutions often address specific subproblems of the dataflow paradigm with varying assumptions, or use scheduling algorithms that are inadequate in search speed and result efficiency [2, 26].

To overcome these issues, we[1] present *comprehensive and pragmatic dataflow representations* for dense NNs, which cover a rich set of existing *temporal* and *spatial* dataflow schemes at both *inter-layer* and *intra-layer* levels, and support both *training* and *inference* workloads. We first illustrate an informal *hierarchical dataflow taxonomy*. This taxonomy reveals the tight coupling across different levels, which hinders us from quick design space exploration. Then, we introduce a set of formal *tensor-centric dataflow directives* that improve over prior work [40]. By treating the tensors stored throughout the accelerator memory hierarchy as first-class citizens, we can directly inspect both the data sizes occupying a buffer and the data access volumes across buffers, which are crucial for determining whether a dataflow scheme is *valid* and *efficient*.

We then build a *generic, optimized, and fast dataflow solver*, KAPLA, which could effectively explore the complicated design space with inter-layer and intra-layer, temporal and spatial dataflow schemes, on both large multi-node accelerators and small edge devices. KAPLA leverages two key techniques to effectively handle *validity check* and *efficiency estimation*, the two main tasks in dataflow solvers. First, KAPLA effectively *decouples the design space*, enabling

[1]A full version of this paper is available at [19].

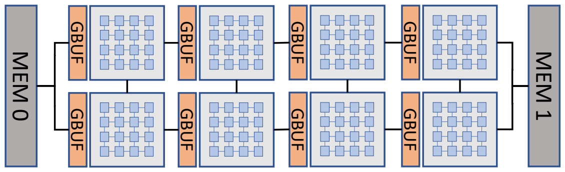


Figure 1: A multi-node NN accelerator. Each node has an array of PEs, a register file per PE, and a global SRAM buffer.

fast upper-level inter-layer pruning with negligible loss in optimality. Second, KAPLA *adopts a novel bottom-up cost descending way* to efficiently solve intra-layer schemes, under a uniform description for multi-level memory hierarchies and supporting advanced data sharing options such as systolic [12] and buffer sharing [7]. KAPLA makes use of our *pragmatic tensor-centric directives* to better capture *spatial resource utilization* and *temporal data movements*, for convenient performance and efficiency evaluation.

Evaluated with a wide range of modern NNs on scalable multi-node accelerators, KAPLA achieves within just 2.2% and 7.7% energy overheads on average compared to the exhaustively searched optimal, for training and inference, respectively. KAPLA can complete such high-quality dataflow scheduling up to two orders of magnitude faster than prior approaches, and typically within *a few minutes* even for large NN training. This is a significant step towards real-time interactive compilation of optimized dataflow on scalable accelerators. We also demonstrate KAPLA is robust to different hardware configurations, including small edge accelerators, and both Eyeriss-like and TPU-like architectures. KAPLA is open-sourced at https://github.com/tsinghua-ideal/kapla.

2 BACKGROUND AND MOTIVATIONS

2.1 Scalable NN Accelerators and Dataflow

Many existing NN accelerators are derived from a common spatial architecture consisted of an array of processing elements (PEs) and a hierarchy of on-chip storage, e.g., a global buffer (GBUF) and per-PE register files (REGF) [3, 12]. As more complex and deeper NNs emerge, recent NN accelerators have to *scale* their compute capabilities through a parallel, multi-node architecture on a single chip or across multiple chips/chiplets [6, 7, 29, 39], as shown in Figure 1, which interconnects a group of nodes and multiple off-chip memories using on-chip and inter-chip networks.

With the abundant compute and buffer resources, finding the best *dataflow schemes* to schedule various NN layers becomes a critical task. Prior research has proposed various optimized dataflow schemes, such as systolic or stationary styles of spatial data transfers in PE arrays [3, 12], loop transformation heuristics to improve data reuse in on-chip buffers [26], workload partitioning and coordination across multiple nodes and chips [6], pipelined processing across layers [7, 39], as well as dataflow specialized for sparse NNs [27]. Our multi-node NN accelerator in Figure 1 supports a rich set of dataflow. A systematic description can be found in Section 3.1.

2.2 Motivations and Challenges

Representing dataflow options. There has been a large body of previous work on understanding and representing NN dataflow [15, 22, 28, 37, 40]. Using generic nested loop structures [26] requires complex and recursive procedures that are hard to reason about. For example, deriving tensor sizes and access traffic at a buffer needs to analyze relevant blocked loops at *all* higher levels in a deep memory hierarchy. The temporal/spatial directives [15, 37] and the Halide/Interstellar scheduling primitives [40] still manipulate individual loop dimensions rather than tensors, and suffer from the same issues. TENET represents flexible and irregular dataflow at the cost of higher complexity to evaluate each scheme [22]. Actually, none of these representations was proposed *together with* a non-trivial search algorithm, which implies that they did not include pragmatic exploration as the primary goal.

Solving optimized dataflow. A good dataflow solver should find *optimized* schemes for a *generic* set of hardware and software configurations, and most importantly, do so sufficiently *fast*. First, in many recent cases dataflow scheduling is not a one-time cost, e.g., network architecture search (NAS) [36, 41]. Second, with the emerging Machine-Learning-as-a-Service (MLaaS) paradigm, if the clients are allowed to supply their own models, fast compilation and dataflow scheduling at the cloud side would be desired. Unfortunately, existing techniques fail to achieve the optimized, generic, and fast requirements simultaneously. A large body of prior work adopts brute-force exhaustive search [3, 7]. While being generic and optimized, such search is quite *slow* with large NNs and multi-node accelerators. Alternatively, heuristic-based algorithms speed up the exploration by either conducting random search [26], or applying machine-learning (ML) methods [2, 9]. These heuristic algorithms are also difficult to rule out "invalid" schedules on specialized accelerators, such as those whose blocked data fail to fit in on-chip buffers [26]. CoSA [11] models dataflow scheduling as mixed-integer programming (MIP) and uses a commercial MIP solver. Although it only takes a few seconds to schedule a single layer, the MIP formulation is unlikely to scale up due to the exponentially growed complexity with the number of variables. Furthermore, most prior efforts just focus on their own sub-problems [3, 6, 7, 11, 26, 32, 40]. The insights from these solvers may be incompatible or even conflicting due to different assumptions for the rest of the design space. Simply combining them will likely lead to suboptimal results.

Table 1: Previous work summarized using our hierarchical dataflow taxonomy.

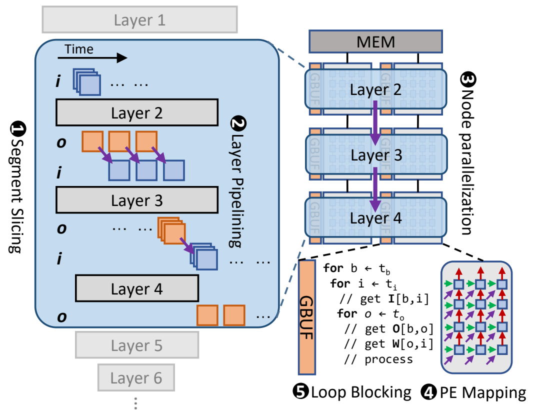
Inter-layer	Temporal	Segment slicing [7, 25, 30]
	Spatial	Layer pipelining [7, 17, 21, 33, 39]
Intra-layer	Spatial	Node parallelization [6, 11, 13, 32, 37]
	Spatial	PE mapping [3, 5, 12, 16, 22, 23, 26]
	Temporal	Loop blocking [1, 2, 6, 9, 11, 26]

3 DATAFLOW REPRESENTATIONS

3.1 A Hierarchical Dataflow Taxonomy

Targeting at both *inter-layer* and *intra-layer* levels, Table 1 summarizes previous proposals into our taxonomy.

Inter-layer dataflow can be categorized into *temporal* and *spatial* scheduling across multiple layers. First, *segment slicing* temporally schedules different groups of consecutive layers (called *segments*) from a large NN on the same accelerator [7] (Figure 2 ❶).



```
for b ← t_b
 for i ← t_i
  // get I[b,i]
  for o ← t_o
   // get O[b,o]
   // get W[o,i]
   // process
```

Figure 2: The hierarchical NN dataflow taxonomy.

Different segments time-share the hardware to utilize the limited resources while all layers within a segment execute together (Figure 2 ❷). Then, *layer pipelining* spatially schedules the layers in a segment onto different nodes of the accelerator with intermediate fmap tensors forwarded on-chip in a pipelined way [33].

Intra-layer dataflow can be described using loop transformations on the layer's nested loops [26, 40]. We focus on the two-level buffer hierarchy (PE REGFs and node GBUFs) in our assumed architecture. At each level, loop unrolling results in *spatial* parallelization, while loop blocking and reordering optimize *temporal* data reuse. At the node level, *node parallelization* spatially parallelizes the layer across nodes (Figure 2 ❸). When a certain dimension is selected, the tensors that contain this dimension are partitioned into smaller subtensors. The other shared tensors need to be replicated in different nodes. To avoid such data duplication, the buffer sharing optimization [7] maintains a single copy of the shared tensor across multiple buffers, and rotates each other's share to allow every node to have a chance of local access at different time. *PE mapping* spatially distributes the computations onto the PE array. The communication of the PE array includes row stationary [3], multicasting [20], and systolic flow [12] (Figure 2 ❹).

Finally, *loop blocking and reordering* have been extensively studied to improve data reuse across the memory hierarchy, by dividing the tensors into smaller blocks and finding the best order to fetch them [26, 40] (Figure 2 ❺). These schemes are difficult to optimize and it is therefore common to use exhaustive, random, or other heuristic search methods [3, 26, 40].

3.2 Tensor-Centric Dataflow Directives

We now propose a formal and quantitative representation with a set of *tensor-centric directives.* The dataflow of the directives is built from the core of the memory hierarchy outward, focusing on spatial resource utilization and temporal data movements. These metrics are essential for assessing the validity, efficiency, and performance of a dataflow scheme. Initially, tensors are allocated and processed on a single PE. This setup is then expanded across the PE array by replicating or sharding tensors with varying offsets across different PEs. Temporal data movements for each iteration are analyzed recursively at each level, including PEs and nodes.

Directive definitions. `tensor(dim=size, ...[, shr])` declares a multi-dimensional (sub)tensor allocated at a certain level of the memory hierarchy, with the given `size` along each `dim`. The optional sharing factor `shr` is used when the tensor is stored across multiple buffers in a shared manner and each buffer only holds `1/shr` of the data [7]. `tensor` declarations are organized in a two-level name scope, across NN layers and across memory hierarchy levels (Listing 1). Within each layer, tensors with the same name but at different memory levels belong to the same full tensor, and tensors in faster levels are subsets of tensors in slower levels. Such naming allows us to infer the detailed tensor *layout* (e.g., tiling strides) in each buffer from the dimension sizes of the tensors in neighboring level buffers. For example, a tensor with `C=3` and `C=6` in two neighboring levels means that it should be tiled by 2.

`stack(dim+=shift, ..., repl)` denotes spatial parallelization where `repl` PEs or nodes exist at a memory hierarchy level, each with a local buffer storing a copy of all tensors declared at this level, potentially offset by `shifts` along `dims`. Tensors are essentially stacked across `repl` buffers, or sharded. `shifts` need not match tensor `sizes`: smaller `shifts` partially overlap tensors, while larger `shifts` distribute non-contiguous ranges. This enables fine-grained interleaving or coarse-grained partitioning. Without `shifts`, tensors are replicated. `stack`s at the same level are applied recursively in order, allowing hybrid and complex spatial parallelism. For instance, two `stack`s in Lines 6 and 7 at the `REGF` level of Listing 1 indicate a 2D PE array mapping.

`update(dim+=step, ...)` denotes ordered, nested temporal iterations in buffers, with tensors in each buffer updated synchronously by increments of `steps` along their dimensions. This process replaces old data with new, and the `step` can either match the tensor shape or support overlapped windows and non-contiguous strides.

An example. Listing 1 shows a dataflow scheme for two adjacent CONV and depthwise CONV layers on a multi-node accelerator. The `CONV:REGF` level represents the row-stationary 2D PE mapping [3]. First, three `tensor`s are allocated in each PE that does a 1D convolution between one row of the filter weights (`R=5,S=1`) and the corresponding range of one input row (`Xi=5,Yi=1`) to calculate one output element (`Xo=Yo=1`). Then the first `update` in Line 8 slides over the entire fmap row by processing a new element along `Xi` and `Xo` each cycle. Data from multiple channels (`C,K`) are stored in the registers to improve reuse. The two `stack`s indicate how data are distributed across the PE array. The first `stack` increments `Yo`, so each PE array column corresponds to a different row of the output tensor. If the fmap tensors have more rows than the number of array columns, the `update` in Line 9 temporally folds the data onto the same physical array. Similarly, the second `stack` increments `S`, resulting in different rows of the filter weight tensor across the PE array rows. `Yi` is incremented in both, making rows of the input tensor distributed diagonally. Finally, the last three `update`s represent the nested loop order.

The `CONV:GBUF` level similarly defines three tensors that will be stored in the node buffer. Here we spatially map the layer onto a region of 4×16 nodes, by using hybrid output and batch parallelization (`K` and `N`) [6]. We enable sharing the input tensors across four node `GBUF`s with the buffer sharing dataflow [7], which is determined by the `repl` of `stack` in Line 17 for output parallelization.

Listing 1: Directive examples for CONV and depthwise CONV layers, with row-stationary PE mapping [3], output + batch hybrid node parallelization [6], and layer pipelining [7].

```
CONV:
 REGF:
  tensor{0}(N=1, C=2, Xi=5, Yi=1)
  tensor{w1}(C=2, K=3, R=5, S=1)
  tensor{1}(N=1, K=3, Xo=1, Yo=1)
  stack(Yi+=1, Yo+=1, 8)  % PE columns
  stack(S+=1, Yi+=1, 5)   % PE rows
  update(Xi+=1, Xo+=1)    % 1D conv
  update(Yi+=8, Yo+=8)    % folding
  update(N+=1)
  update(C+=2)
  update(K+=3)
 GBUF:
  tensor{0}(N=4, C=4, Xi=19, Yi=19, shr=4)
  tensor{w1}(C=4, K=6, R=5, S=5)
  tensor{1}(N=4, K=6, Xo=15, Yo=15)
  stack(K+=6, 4)    % output node parallel
  stack(N+=4, 16)   % batch node parallel
  update(C+=4)
  update(K+=24)
  update(N+=64)
DWCONV:
 REGF:
  % ...
 GBUF:
  % DWCONV input is the same as CONV output
  tensor{1}(N=4, C=4, Xi=9, Yi=15)
  tensor{w2}(C=4, R=3, S=3)
  tensor{2}(N=4, C=4, Xo=4, Yo=7)
  stack(C+=4, 6)    % channel node parallel
  stack(N+=4, 16)   % batch node parallel
  stack(Xo+=4, 2)   % output width node parallel
  update(Yo+=7)
  update(C+=24)
  update(N+=64)
```

The DWCONV section shows how the directives support a depthwise CONV as non-CONV/FC layers. The key difference from the previous CONV is that C and K are the same dimension now, so **tensor**{w2} only has C,R,S, and the output **tensor**{2} uses the same C=4 as the input **tensor**{1}. It also uses a more complex parallelization scheme across 12×16 nodes, where channel C, batch N, and output width Xo are unrolled by 6, 16, and 2, respectively.

Finally, we notice that the input tensor of DWCONV is the same as the output tensor of CONV, i.e., **tensor**{1}, indicating inter-layer data dependencies. Their sizes at the GBUF level are the same, e.g., K=6 stacked by 4 vs. C=4 stacked by 6. Their top-level loop blocking schemes also match, with **update**(K+=24) vs. **update**(C+=24), and both **update**(N+=64). Consequently, they are produced and consumed in the same way by the two layers, forming a fine-grained on-chip forwarding pipeline [7].

Advantages. First, by treating tensors as first-class citizens using **tensor**s and explicitly describing the changes of the tensors over time, we can directly inspect the blocked data sizes allocated in every buffer in each memory level, and easily calculate the amount of data transfers across the memory hierarchy. This would however require complex algorithms in previous frameworks like MAESTRO [15] and Interstellar [40] (Section 2.2). Second, our directives can represent various inter-layer pipelining schemes using matched **tensor** names for data dependencies and matched **update**s for transfer granularities. These are difficult with MAESTRO [15] or Interstellar [40], especially for complicated DAGs.

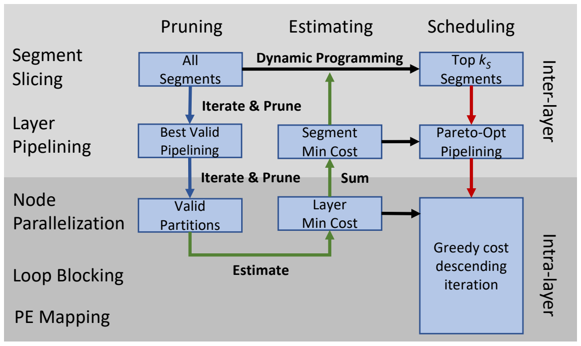


Figure 3: The overall workflow of KAPLA. The blue, green, and red arrow chains represent the pruning, estimating, and scheduling processes, respectively.

4 KAPLA DATAFLOW SOLVER

In this section, we present a generic, optimized, and fast dataflow solver, KAPLA, that outperforms existing random and ML-based search. KAPLA is implemented in Python and open-sourced.

4.1 High-Level Ideas

Dataflow solvers must effectively handle: (1) *validity check*, i.e., how to quickly determine whether a scheme satisfies all constraints. (2) *efficiency estimation*, i.e., how to accurately estimate the performance/energy of a scheme. Figure 3 shows the workflow of KAPLA. Initially, we examine dataflow schemes at inter-layer levels, performing a quick, conservative evaluation of the necessary computation and buffer resources to effectively prune invalid schemes. We then rank the remaining viable schemes based on their costs, selecting the top candidates for further intra-layer examination through a greedy cost descending algorithm. KAPLA employs novel search strategies built on the pragmatic tensor-centric directives discussed in Section 3.2, which enhance the assessment of spatial resource utilization and temporal data movements. KAPLA supports advanced and complex data sharing options such as multicast [3], systolic [12], and buffer sharing [7], by uniformly modeling the same-level and next-level data transfers.

4.2 Inter-Layer Pruning and Prioritization

As detailed in Section 3.1, KAPLA manages temporal and spatial resource allocation through segment slicing and layer pipelining. For quick validity checks, it relies on *conservative pruning* to narrow down feasible designs. For efficiency, it uses *fast cost estimation* based on dynamic programming to identify and prioritize promising schemes for detailed analysis during the intra-layer phase.

Conservative validity pruning. KAPLA conducts fast and conservative pruning at the inter-layer level, by directly estimating the minimum required buffer capacity of a layer at each memory level, using the best knowledge it has at this point. If the buffer capacity is even smaller than this minimum requirement, the layer is guaranteed to not fit with any possible intra-layer scheme. Therefore such pruning is fast (not exploring detailed intra-layer schemes) and conservative (never rejecting valid inter-layer schemes). Nevertheless, the pruning is quite effective in practice, able to eliminate over 90% candidates and result in 10× search speedups in our evaluation.

Specifically, KAPLA assesses whether a layer's data can fit within the total buffer capacity across all nodes assigned to it, without considering the finer details of intra-layer spatial and temporal mapping. This evaluation is expedited by focusing only on the outermost buffer level, and remains conservative due to potential increases in the required capacity from indivisible partitioning or tensor replication across nodes. However, since the data dimensions usually far exceed the number of nodes and some mechanisms like buffer sharing [7] can reduce data duplication, false positives are rare, leading to high pruning effectiveness.

Fast cost estimation. To prioritize potentially optimized schemes, KAPLA assigns a cost to each inter-layer scheme of every segment. The key insight is to approximate towards optimistic cases when information is insufficient, making the estimated cost a relatively tight lower bound. When used to *prioritize* candidates, a lower bound effectively reflects the scheme's potential, and our prioritization method described below can tolerate small errors.

For example, while DRAM and inter-node access volumes are determined only by the top buffer level dataflow, node buffer accesses depend on inner levels between PEs and nodes. Exploring these levels enlarges the design space and slows down the search, so we use the minimum achievable access count. For resource utilization, we assume the layer could use *all* PEs across all assigned nodes, temporarily ignoring internal fragmentation. For layer pipelining, if the top-level inter-layer scheme meets the fine-grained forwarding requirements (e.g., matched data forwarding granularities at the top level), we enable it when summing the total pipeline execution time, i.e., waiting for one fmap instead of all [7].

During the cost estimation, KAPLA includes another pruning pass, skipping schemes with non-Pareto-optimal access counts among the multiple tensors.

Use of directives. To explore only the upper-level schemes, we target only the topmost GBUF level directives without looking at the internal **tensor**s and their **stack** or **update** operations. The upper-level access counts, particularly DRAM and inter-node, are mostly determined by the top-level (GBUF) **tensor** and **update** directives, and can be directly calculated. Some of these directive parameters are constrained. For example, adjacent layers must have equal **tensor** sizes and matched **update** steps to use on-chip pipelined forwarding. KAPLA recognizes these constraints.

Dynamic-programming-based prioritization. We have now selected a few potentially optimized inter-layer schemes for each segment, along with their estimated costs. Next, to explore segment slicing and obtain the best overall segment chains for the entire NN, KAPLA uses dynamic programming. It processes each layer in the DAG topological order, finding at each step the segment chain that *ends at* the current layer with the minimum aggregated cost. With accurate cost estimation, dynamic programming provides optimized results in linear time. To accommodate potential errors, instead of a single best segment chain, KAPLA keeps the top k_S candidates. We use a default k_S of 4 and evaluate its impact in Section 6.2.

4.3 Intra-Layer Stacking and Caching

With the short list of potentially optimized inter-layer schemes from Section 4.2, KAPLA finalizes intra-layer scheduling. For a multi-level memory hierarchy, KAPLA works through each level

Algorithm 1 Greedy cost descending algorithm.

```
1:  Input: NN layer specification LayerSpec, hardware specification
    HWSpec, and constraints Constrs
2:  Output: Optimized dataflow scheme OptDf
3:  MemoryHier = [RegFile, GBUF, DRAM]
4:  Df, LoopCnt = initUnitMapping(LayerSpec, HWSpec)
5:  for each m in MemoryHier do
6:      for order in genLoopOrder(m) do
7:          Df, LoopCnt = stack(Df, m, LoopCnt, order)
8:          Df, LoopCnt = cache(Df, m, LoopCnt, order)
9:          if checkValid(Constrs, LoopCnt) is False then
10:             continue
11:         if evalCost(Df) > evalCost(OptDf) then
12:             OptDf = Df
```

sequentially, from smaller, near-level buffers (e.g., REGF) to larger, far-level ones (e.g., GBUF). This bottom-up approach matches common dataflow optimization practices, i.e., first maximizing the use of faster, cheaper buffers, then moving to slower, more expensive levels. This ensures buffer capacity and parallelism constraints are always satisfied. In contrast, traditional loop blocking approaches factorize loop factors top-down, producing many invalid schemes and requiring extensive validity checks.

Within each memory level, KAPLA maximizes the *sizes* of tensors stored and minimizes the *accesses* to the next level. Starting from the smallest *unit tensors* matching the previous level's sizes (determined by the PE computation patterns), two optimization passes—*stacking* and *caching*—enlarge the unit tensors along the selected dimensions until the buffer capacity is fully used or the tensors are fully stored. Stacking parallelizes tensors across buffers, while caching increases tensor sizes in each buffer, both reducing next-level accesses. We prioritize stacking over caching to improve parallelism.

The selected tensor dimensions for stacking and caching must balance the reuse of multiple tensors, a non-convex problem difficult to solve efficiently [40]. We use a greedy, iterative *cost descending* method. At each step, we calculate the access count for each tensor and choose a dimension that helps the tensor with the maximum access count, breaking ties with the second most accessed tensor. The chosen dimension is then slightly enlarged. We continue such iterations until the buffer capacity is used up. Our tensor-centric directives enable fast capacity checks and access count calculations.

Algorithm 1 summarizes the intra-layer greedy cost descending algorithm. We first initialize the unit tensor sizes and the dataflow required by the specific hardware (e.g., systolic arrays) and the NN layer. For each memory level bottom-up along the hierarchy, we iterate over different loop orders and apply stacking and caching. Then we check various constraints and update the current best scheme at this memory level.

Figure 4 shows an example. We use **stack**s and enlarge **tensor** dimension sizes for stacking and caching, respectively, from the inside out along the memory hierarchy. We can first decide the directive parameters at inner levels, and then freely select the dimensions to enlarge or stack at outer levels. Initially, a unit tensor of size 1 is constructed. In the stacking pass, we select dimensions K and C to use 8×4 buffers, each storing a tensor with the original size of 1. The remaining factors along these dimensions are correspondingly reduced. Then in the caching pass, we select dimensions N

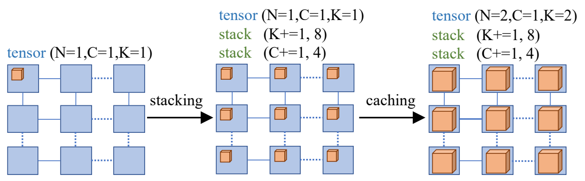


Figure 4: An example of intra-layer stacking and caching.

and K, and enlarge the stored tensors to a size of N=2,C=1,K=2. The next memory level will start from a unit tensor with this size, and work with the remaining dimension factors.

5 METHODOLOGY

Evaluated workloads. We evaluate both inference and training on five CNNs (including AlexNet [14], MobileNet [10], VGGNet [31], GoogLeNet [35], ResNet [8]), an MLP [4], and an LSTM [34], with a default batch size of 64 for large multi-node accelerators and 1 for small single-node edge devices.

Hardware architectures. Our target architecture (Figure 1) consists of 16 × 16 nodes, each of which contains an 8 × 8 PE array with a 64 B register file per PE, and a 32 kB global buffer [7]. We model it in 28 nm and assume a 500 MHz logic frequency. The PE array uses Eyeriss-like row-stationary mapping [3]. The 16-bit MAC in a PE costs 1 pJ [3, 5, 23]. McPAT 1.3 models the register files and SRAM buffers of different sizes, as well as the PE array buses [18]. The NoC energy is 0.61 pJ/bit per each hop [38]. The off-chip memory supplies 25.6 GBps bandwidth with four LPDDR4 channels, with energy modeled from commercial datasheets [24]. Besides the large multi-node accelerator, we also evaluate a small single-node edge inference device. It contains a 16×16 PE array with 512 B registers per PE, and a 256 kB global buffer. We intentionally use a different, TPU-like systolic array [12] to demonstrate the generality of KAPLA. We use the nn-dataflow simulator [6, 7] to model the above two architectures. This simulator is validated against both cycle-accurate simulation and real Eyeriss results [6]. We extend it to support training and systolic arrays.

Baseline solvers. We compare KAPLA (K) with state-of-the-art scheduling approaches. The baseline (B) is nn-dataflow [7] which uses exhaustive search. We implement another exhaustive search (S) using our directives. The random search (R) from Timeloop [26] evaluates candidates at each level with an empirical probability. We set it to be no less than 0.1. The ML-based method (M) from AutoTVM [2] uses simulated annealing guided by XGBoost.

6 EVALUATION

6.1 Dataflow Efficiency and Scheduling Speed

For training on the large multi-node architecture, Figure 5 illustrates the energy and the execution time of the resulted dataflow schedules from the five scheduling approaches, normalized to the baseline B. The energy and performance results generally follow the same trends. First, exhaustively searching on our tensor-centric directives (S) is able to match the baseline efficiency, demonstrating the generality of the directives. In some cases like MobileNet and MLP, S even results in slightly better schemes, because the directives

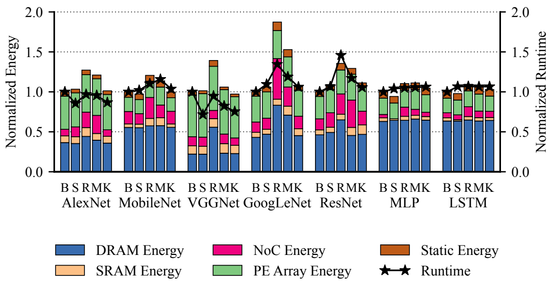


Figure 5: Comparison of dataflow energy and performance for training on multi-node Eyeriss-like accelerators with batch 64.

Table 2: Comparison of scheduling time for training on multi-node accelerators. Measured on an Intel Xeon Gold 5120 processor with 8 parallel Python processes.

	B	S	R	M	K
AlexNet	8.7 h	12.7 h	4.6 min	3.2 h	32 s
MobileNet	17.6 h	33.8 h	16.6 min	10.1 h	53 s
VGGNet	6.8 h	12.2 h	5.5 min	3.6 h	35 s
GoogLeNet	64.2 h	111.0 h	31.4 min	37.1 h	262 s
ResNet	78.6 h	114.8 h	42.4 min	28.6 h	118 s
MLP	0.2 h	0.7 h	1.0 min	0.2 h	13 s
LSTM	0.2 h	0.9 h	7.4 min	0.1 h	6 s

are more flexible and support a larger design space than prior work. Second, random search R is the least robust method, whose results vary significantly across different NNs, with up to 1.9× worse energy. This demonstrates that randomly skipping candidates without considering the design space characteristics is insufficient. Third, the ML-based M works better than R, with 17.7% on average and up to 1.5× worse results than optimal. We suspect that complex NNs like GoogLeNet and ResNet have larger design spaces and more constraints, which are hard for ML algorithms to learn.

Compared to the exhaustively searched optimal, KAPLA only has a 2.2% energy overhead on average, and within 10% for the worst case MLP. Furthermore, the energy breakdowns across major hardware components also match well. As different components are mainly affected by different scheduling levels, the results indicate that the design space decoupling of KAPLA works well and each level's solver is individually optimized.

For scheduling speed in Table 2, KAPLA significantly outperforms all alternative approaches. Compared to the baseline B, KAPLA offers 518× speedup on average. It is also 265× faster than ML-based M, as the ML-based method needs to iterate for many steps to converge. The more complex an NN is, the more speedup KAPLA can achieve. While the other scheduling methods spend hours or days, KAPLA can typically output optimized schemes within a few seconds and up to a few minutes. In summary, KAPLA realizes a more practical approach towards real-time interactive compilation of optimized dataflow on large-scale NN accelerators.

We also evaluate inference on the multi-node architecture in Figure 6. Without back-propagation layers, the inference DAGs

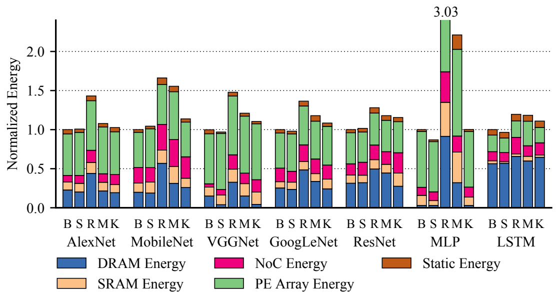


Figure 6: Comparison of dataflow energy for inference on multi-node Eyeriss-like accelerators with batch 64.

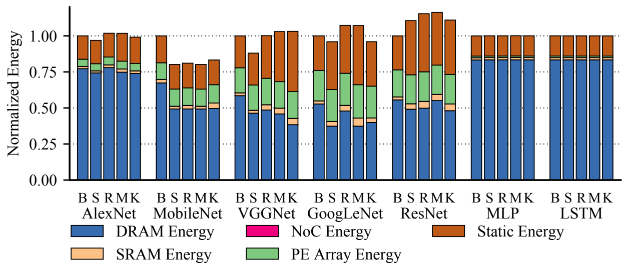


Figure 7: Comparison of dataflow energy for inference on single-node TPU-like accelerators with batch 1.

are much simpler than the training DAGs and have fewer constraints [7]. Therefore the scheduling becomes more involved with more options. Nevertheless, KAPLA has only a 7.7% energy overhead on average compared to B, but is 174× faster. It is also 50× and 1.5× faster than M and R, while the latter two have much worse average efficiency drops of 36.1% and 59%, respectively. Notice that the performances of ML-based and random-based methods are poor on MLP. We attribute this to the low computation-storage ratio of MLP, which further leads to tight storage constraints and hinders the discovery of good or even valid dataflows of the two methods.

Figure 7 further demonstrates KAPLA's generality on a small edge accelerator with only a 256 kB buffer, and a TPU-like systolic array. Since the design space is small, all methods can achieve near-optimal efficiency. The ML-based method has an overhead up to 16%. We have to set the random search probability to 0.85 to get valid schemes under the rigid constraints of a small on-chip buffer, which leads to an average of 3.8% energy overhead. This indicates that ML- and random-based approaches suffer from generality issues in setting their hyperparameters. KAPLA introduces a 1.9% overhead on average, with the worst case 10% on ResNet. This shows that KAPLA is robust even in resource-limited scenarios.

6.2 Sensitivity Studies

Figure 7 shows KAPLA works with different PE array architectures. Table 3 further shows how KAPLA behaves on more hardware configurations. The overheads are generally small, demonstrating that KAPLA is robust to hardware changes.

Table 3: Energy overheads of the KAPLA dataflow results for GoogLeNet with different configurations.

Batch	Nodes	PEs	GBUF	REGF	Overhead
64	4 × 4	8 × 8	32 kB	32 B	4.4%
64	4 × 4	8 × 8	32 kB	64 B	2.7%
64	4 × 4	8 × 8	32 kB	128 B	1.5%
8	4 × 4	16 × 16	32 kB	32 B	8.3%
1	16 × 16	8 × 8	32 kB	64 B	6.7%

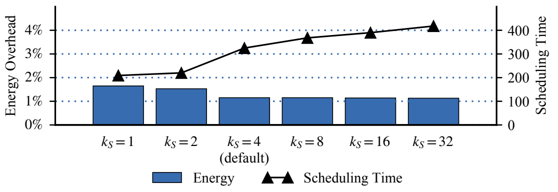


Figure 8: Impact of segment candidate counts in KAPLA.

KAPLA's dynamic-programming-based prioritization tolerates cost estimation errors by keeping up to k_S candidate segments. The default value is 4. We study its impact on the tradeoff between dataflow efficiency and scheduling time, as shown in Figure 8. We see that as k_S decreases, the energy overheads do not increase much, meaning that the cost estimation errors are insignificant. However the search speed substantially improves.

7 CONCLUSIONS

In this paper we present a hierarchical taxonomy and a set of tensor-centric directives to describe the rich NN dataflow space. We then build a generic, optimized, and fast dataflow solver, KAPLA, and demonstrate its effectiveness when scheduling complex NNs on various scales of accelerators for training and inference. KAPLA achieves near-optimal schedules, and is significantly faster than previous methods.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. Mingyu Gao is the corresponding author.

REFERENCES

[1] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation. In *8th International Conference on Learning Representations (ICLR)*.
[2] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *32nd International Conference on Neural Information Processing Systems (NeurIPS)*. 3389–3400.
[3] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *43rd International Symposium on Computer Architecture (ISCA)*. 367–379.
[4] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *43rd International Symposium on Computer Architecture (ISCA)*. 27–39.
[5] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting

Vision Processing Closer to the Sensor. In *42nd International Symposium on Computer Architecture (ISCA)*. 92–104.

[6] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 751–764.

[7] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 807–820.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[9] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W Fletcher. 2021. Mind Mappings: Enabling Efficient Algorithm-Accelerator Mapping Space Search. In *26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 943–958.

[10] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861* (2017).

[11] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. 2021. CoSA: Scheduling by Constrained Optimization for Spatial Accelerators. In *48th International Symposium on Computer Architecture (ISCA)*. 554–566.

[12] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *44th International Symposium on Computer Architecture (ISCA)*. 1–12.

[13] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In *43rd International Symposium on Computer Architecture (ISCA)*. 380–392.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *26th International Conference on Neural Information Processing Systems (NeurIPS)*. 1097–1105.

[15] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach. In *52nd International Symposium on Microarchitecture (MICRO)*. 754–768.

[16] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 461–475.

[17] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. 2016. A High Performance FPGA-based Accelerator for Large-Scale Convolutional Neural Networks. In *26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–9.

[18] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *42nd International Symposium on Microarchitecture (MICRO)*. 469–480.

[19] Zhiyao Li and Mingyu Gao. 2023. KAPLA: Pragmatic Representation and Fast Solving of Scalable NN Accelerator Dataflow. *arXiv preprint arXiv:2306.15676* (2023).

[20] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An Instruction Set Architecture for Neural Networks. In *43rd International Symposium on Computer Architecture (ISCA)*. 393–405.

[21] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing CNN Model Inference on CPUs. In *2019 USENIX Annual Technical Conference (ATC)*. 1025–1040.

[22] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. 2021. TENET: A Framework for Modeling Tensor Dataflow Based on Relation-Centric Notation. In *48th International Symposium on Computer Architecture (ISCA)*. 720–733.

[23] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks. In *23rd International Symposium on High Performance Computer Architecture (HPCA)*. 553–564.

[24] Micron Technology Inc. 2014. Mobile LPDDR4 SDRAM: 272b: x64 Mobile LPDDR4 SDRAM Features.

[25] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. 2018. A Hierarchical Model for Device Placement. In *International Conference on Learning Representations (ICLR)*.

[26] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315.

[27] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks. In *44th International Symposium on Computer Architecture (ISCA)*. 27–40.

[28] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 519–530.

[29] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *52nd International Symposium on Microarchitecture (MICRO)*. 14–27.

[30] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. In *44th International Symposium on Computer Architecture (ISCA)*. 535–547.

[31] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[32] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2020. AccPar: Tensor Partitioning for Heterogeneous Deep Learning Accelerators. In *26th International Symposium on High Performance Computer Architecture (HPCA)*. 342–355.

[33] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *23rd International Symposium on High Performance Computer Architecture (HPCA)*. 541–552.

[34] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to Sequence Learning with Neural Networks. In *28th International Conference on Neural Information Processing Systems (NeurIPS)*. 3104–3112.

[35] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–9.

[36] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. 2019. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *2019 Conference on Computer Vision and Pattern Recognition (CVPR)*. 2815–2823.

[37] Zhanhong Tan, Hongyu Cai, Runpei Dong, and Kaisheng Ma. 2021. NN-Baton: DNN Workload Orchestration and Chiplet Granularity Exploration for Multichip Accelerators. In *48th International Symposium on Computer Architecture (ISCA)*. 1013–1026.

[38] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Software-Defined Cache Hierarchies. In *44th International Symposium on Computer Architecture (ISCA)*. 652–665.

[39] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. 2017. ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks. In *44th International Symposium on Computer Architecture (ISCA)*. 13–26.

[40] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 369–383.

[41] Barret Zoph and Quoc V. Le. 2016. Neural Architecture Search with Reinforcement Learning. *arXiv preprint arXiv:1611.01578* (2016).