2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)

Timeloop: A Systematic Approach to DNN Accelerator Evaluation

Angshuman Parashar* Priyanka Raina*‡ Yakun Sophia Shao* Yu-Hsin Chen*
Victor A. Ying† Anurag Mukkara† Rangharajan Venkatesan* Brucek Khailany*
Stephen W. Keckler* Joel Emer*†

*NVIDIA †Massachusetts Institute of Technology ‡Stanford University

***Abstract*—This paper presents Timeloop, an infrastructure for evaluating and exploring the architecture design space of deep neural network (DNN) accelerators. Timeloop uses a concise and unified representation of the key architecture and implementation attributes of DNN accelerators to describe a broad space of hardware topologies. It can then emulate those topologies to generate an accurate projection of performance and energy efficiency for a DNN workload through a mapper that finds the best way to schedule operations and stage data on the specified architecture. This enables fair comparisons across different architectures and makes DNN accelerator design more systematic. This paper describes Timeloop's underlying models and algorithms in detail and shows results from case studies enabled by Timeloop, which provide interesting insights into the current state of DNN architecture design. In particular, they reveal that dataflow and memory hierarchy co-design plays a critical role in optimizing energy efficiency. Also, there is currently still not a single architecture that achieves the best performance and energy efficiency across a diverse set of workloads due to flexibility and efficiency trade-offs. These results provide inspiration into possible directions for DNN accelerator research.**

***Index Terms*—modeling; accelerator architecture; deep neural networks; neural network dataflows**

I. Introduction

Deep neural networks (DNNs) have emerged as a key approach for solving complex problems across a wide range of domains, including image recognition [21], speech processing [3], [14], [16], natural language processing [9], language translation [11], and autonomous navigation [23]. To provide high performance and energy efficiency to this class of compute-intensive algorithms, many DNN hardware accelerators have been proposed [1], [2], [5], [6], [12], [13], [15], [18], [25], [29], [33], [35], [39]. While they share similar hardware constructs such as a specialized memory hierarchy [30] and an array of multiply-accumulate units (MACs), they differ widely in their underlying *dataflows*, which define how operations are allowed to be partitioned and scheduled for computation, and how data can be staged in the memory hierarchy [7]. Dataflows have been shown to have a significant impact on performance and energy efficiency due to the resulting differences in data reuse and hardware utilization [6], [19], [22], [25], [31].

The vast number of possible dataflows and the accompanying choices of hardware implementation have created a large *architecture design space*. In addition, for a given DNN layer, i.e., a *workload*, a flexible architecture may permit many different ways of scheduling operations and staging data on the same architecture, which we call different *mappings* [7], resulting in widely varying performance and energy efficiency. This results in a requirement to search for a good mapping within the space of valid mappings, i.e., the *mapspace*, to fairly characterize the architecture. The need to do this search for each workload compounds the complexity of exploring the architecture design space. Prior work on DNN accelerators have dealt with this problem only within the scope of their specific designs [6], [13], [22], [26], [27], [32], [35]–[37]. Tools for broader design space exploration [19], [27], [34], [38] also struggle with challenges such as separation of the architecture design space from the mapspace, enumeration of mappings in the mapspace and quality of performance and energy efficiency evaluation, making it difficult to explore and compare a wide range of architectures.

To address these challenges, we propose *Timeloop*, an infrastructure for evaluating and exploring the architecture design space of DNN accelerators. Timeloop has two main components: a *model* to provide performance, area and energy projections and a *mapper* to construct and search through the mapspace of any given workload on the targeted architecture.

To perform an analysis using Timeloop, a user describes an architecture's organization in a highly configurable template with abstractions for compute units (with configurable precision), memories (in a configurable multilevel hierarchy), and communication links. Architectural *constraints*, including the dataflow and additional hardware attributes such as utilization and bandwidth limitations, can be further imposed to define the mapspace. Given a workload and an architecture specification (i.e., organization and constraints), Timeloop's mapper systematically constructs the mapspace, evaluates the quality of each mapping in the mapspace with its embedded cost model, and searches through the mapspace iteratively for an optimal mapping. This search is feasible thanks to the model's speed (by exploiting the regular structure of DNN workloads) and accuracy (for energy and performance), which has been validated against existing designs.

Timeloop makes the following key contributions:

1) Timeloop provides a concise and unified way of describing the key attributes of a diverse class of DNN architectures and their implementation features as the

input to a fast and accurate analytical model. This allows for the exploration of a wide range of architectures.

2) Timeloop is the first infrastructure to effectively combine the exploration of a large design space with a mapper that allows for finding the optimal mapping of any workload on the targeted architecture. This enables fair comparisons between different architectures and makes the DNN accelerator design more systematic and less of an art.
3) Timeloop enables a number of studies that provide interesting insights into the current state of DNN accelerator architecture design. These studies point out the pros and cons of various representative architectures in the literature and provide inspiration for future research into the design of DNN accelerators.

We expect to release Timeloop under an open-source license and welcome contributions from the community.

II. Motivation

The architectural design of DNN accelerators tends to be more of an art than science due to the large design space and the lack of a systematic approach to explore it. To explore such a design space, we have observed a unique symbiosis between the need for a model that can describe and emulate a wide variety of architectures and the need for a mapper that can optimally configure the accelerator to get a fair characterization of the performance and energy efficiency of a specific workload. In this section, we will describe the symbiosis that Timeloop exploits to make it an effective infrastructure for exploring the design space of DNN accelerators.

A model needs a mapper. Unlike traditional architectures that use an ISA as the hardware/software interface, each DNN accelerator uniquely exposes many configurable settings in the hardware, and its behavior is critically dependent on those settings. Therefore, instead of using pre-compiled binaries or traces, running a DNN workload on an accelerator involves finding a mapping (and corresponding set of configuration settings), which dictates the scheduling of operations and data movement, to maximize performance and energy efficiency.

To obtain a fair characterization of the hardware, it is important to select a good mapping. Figure 1 shows a distribution of the energy efficiency of various mappings for the convolutional layer VGG_conv3_2 on a 1024-MAC architecture similar to NVDLA [28]. The 480k mappings shown here are all within 5% of the peak performance, but vary nearly 19× in energy efficiency. Only one mapping is energy-optimal and 9 others are within 1% of the optimal mapping. The variations arise due to different tiling and scheduling alternatives represented by the mappings, with the more efficient mappings making better use of buffer capacity, network multicast and loop ordering to maximize reuse. However, the optimal mapping can and will likely change across workloads; an optimal mapping for one architecture may also be a poor (or even invalid) choice on another architecture. Therefore, mapping is an essential step in evaluating a DNN workload on an architecture. We show in Section V how Timeloop systematically constructs a mapspace and walks through the mappings in it.

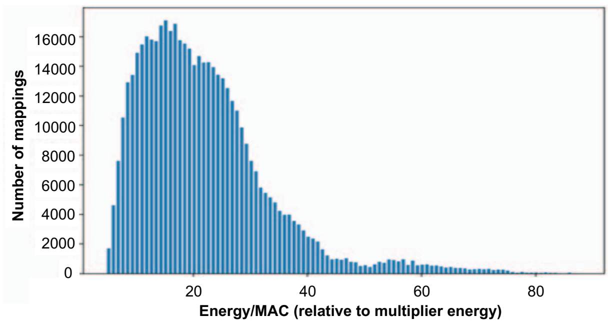


Fig. 1. Histogram of the energy efficiency of various mappings of VGG_conv3_2 on an example architecture.

A mapper needs a model. A mapper enumerates the mapspace and then iteratively searches through the space for the optimal mapping. Thus, it needs a cost function to determine whether one mapping is better than another. This cost function must be *fast* enough to evaluate to allow the mapper to explore a large mapspace in a reasonable amount of time, but it must also be *accurate*, i.e., it must be representative of the actual behavior of the mapping on the architecture, otherwise the mapper will give misleading results. For example, while DRAM accesses consume a lot of energy, accesses to on-chip buffers are often just as expensive. In Figure 1, 6,582 mappings have the same minimum DRAM accesses, but vary 11× in energy efficiency. Even though the actual optimal mapping in this example does have the minimum number of DRAM accesses, it is not guaranteed to be this case across all architectures and technologies. As another example, we show in Section VIII-B that cost ratio changes due to technology trends lead to different optimal mappings, and that the optimal mapping in one technology may be sub-optimal in another.

III. Prior work

There is an abundance of excellent recent work on DNN accelerators [6], [13], [22], [26], [27], [32], [35]–[37]. Across these works, a large component of the mapping is fixed in hardware with a limited set of loop unrolling (parallelism), loop tiling and ordering (data reuse) choices exposed to a search algorithm. Thus, they do provide mappers, but the respective mapspace as well as architectural model are closely tied to the specific attributes of each design and, therefore, are unsuitable for broader design space exploration or competitive assessment.

A general framework for studying a broad range of designs must be able to describe the organization of each design, automatically generate the complete mapspace for any described architecture, and find efficient mappings within the space. Otherwise, a comparison between competing architectures is meaningless. A few recent proposals have attempted this challenging task [19], [27], [34], [38]. Unfortunately, we believe some of them suffer from inadequate architectural cost models [31], [34], [38], with simplistic hardware templates and/or inability to infer different forms of spatial or temporal

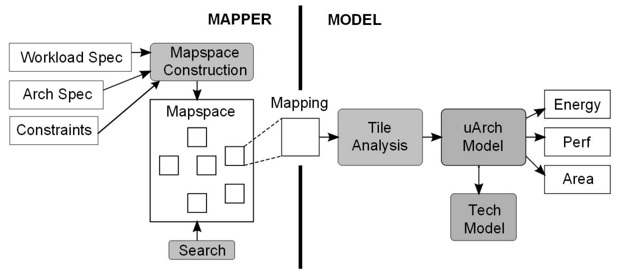


Fig. 2. Timeloop tool-flow.

```
for r=[0:R):
 for s=[0:S):
  for p=[0:P):
   for q=[0:Q):
    for c=[0:C):
     for k=[0:K):
      for n=[0:N):
       Output[p][q][k][n] +=
         Weight[r][s][k][c] * Input[p+r][q+s][c][n];
```

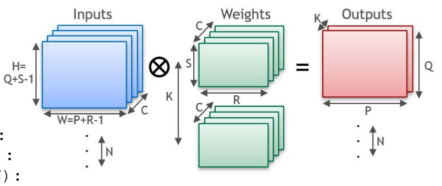


Fig. 3. Convolutional layer 7D loop nest.

reuse. Others fail to comprehend the complete size and scope of the mapspace and/or obfuscate the separation between the architecture design space and mapspace [19], [27]. Finally, recent work on sophisticated compilers for DNN workloads [4], [31] have built-in cost models, though the objective of those models is simply to provide enough fidelity to guide the optimizer rather than to provide accurate energy and performance projections for architecture design-space exploration.

Timeloop provides a rich and general template that is capable of representing DNN accelerators in a large design space, and it automatically constructs the complete mapspace for any specified architecture. We show that popular *dataflows* such as *output-stationary* or *weight-stationary* [6], [8] are but specific instances of a larger set of constraints that can be imposed on the mapspace, limiting the computation schedules and operand reuse patterns that the architecture can exploit. For each workload, Timeloop automatically searches for optimal mappings within these constraints, ultimately providing performance, energy and area characterizations using a detailed architectural cost model. Thus, it aims to serve as a super-set of many of these prior approaches, and as a robust tool for rapid early-stage design space exploration and competitive evaluation of DNN accelerator architectures.

IV. Timeloop Overview

Timeloop's operation consists of creating a mapspace for a given workload on an architecture, exploring that mapspace to find the optimal mapping, and reporting the performance and energy metrics of that optimal mapping. Timeloop needs the following inputs to generate a mapspace:

1) The shape and parameterization of a workload, e.g., the dimensions of the input, output, and weight tensors used in a DNN layer, and the set of operations needed to compute the layer.
2) A specification of the architecture's hardware organization (arithmetic units, memories and networks).
3) The constraints that the architecture imposes on ways in which the computation can be partitioned across the hardware substrate and scheduled over time.

Once the mapspace is constructed, Timeloop evaluates the performance and energy efficiency of a set of mappings within the space. This evaluation does not rely on a cycle-accurate simulator; instead, Timeloop exploits the fact that computation and data-movement patterns in DNN computations are largely deterministic, enabling it to compute throughput and access counts analytically for an architecture. The model determines the counts of various activities, including arithmetic operations, memory accesses and network transfers, including the multi-casting of data from a producer to multiple consumers, and forwarding of data between units. These access counts are used to determine performance. Combined with the energy cost per access from the energy model, they are also used to determine the energy consumption of the workload.

Figure 2 shows an overview of Timeloop. The infrastructure is split into two main components, the mapper and the model, with a mapping serving as the interface between the two. We describe the mapper in Section V along with the input specifications used to configure Timeloop to represent different architectures and workloads. In Section VI, we describe how the model evaluates each mapping.

V. Timeloop Inputs and Mapper

A. Workload Specification

Timeloop's workload format is similar to the form of a single DNN layer. To evaluate a complete network, one can invoke Timeloop sequentially on each layer and accumulate the results. Each layer has tremendous reuse opportunities, and we leave exploration of cross-layer reuse to future work. In this paper, we focus our evaluation efforts on convolutional (CONV) layers shown in Figure 3. These layers can be described as a 7D nested loop over weight tensor's height and width (R, S), output tensor's height and width (P, Q), number of input channels (C), number of output channels (K), and number of inputs or batch size (N). Matrix-matrix multiplications can be expressed as convolutions by setting R, S, P and Q to 1, and matrix-vector multiplications can also be expressed by setting R, S, P, Q and N to 1. Therefore, Timeloop also supports fully-connected (FC) layers and recurrent neural network (RNN) layers since they are essentially matrix-vector multiplications. CONV and FC layers account for a majority of the computation in DNNs (e.g., 99.25% in ResNet50 [17] inference on ImageNet). In general, Timeloop can analyze any workload that can be described as a deep loop nest with fixed base and bounds, with operand indexing expressions that are a linear composition of loop indices and body iterations that may be freely re-ordered.

From the nested loop in Figure 3, we see that loops have constant base and bound. Convolution operand tensors (i.e., weights and inputs) and result tensors (i.e., outputs) are indexed using linear combinations of loop indices, and all loop

```
arch = {                    {
  arithmetic = {              name = "GBuf";
    instances = 256;          sizeKB = 128;
    word-bits = 16;           instances = 1;
  };                          word-bits = 16;
  storage = ({              }, {
    name = "RFile";           name = "DRAM";
    entries = 256;            technology = "DRAM";
    instances = 256;          instances = 1;
    meshX = 16;               word-bits = 16;
    word-bits = 16; },      } ); };
```

Fig. 4. Eyeriss [6] organization described in Timeloop. In addition to this organization spec, mapspace constraints (in Section V-D) are required to capture Eyeriss' row-stationary dataflow.

iterations are commutative. Each iteration of the loop body is a MAC operation, which we refer to as a *point* in the *operation space* of a workload. This space consists of all the integer lattice points enclosed by the 7D hyper-rectangle defined by the loop bounds. These dimensions also define 4D hyper-rectangles for each of the operand and result tensors, which we call *dataspaces*: the $C*K*R*S$ weight tensor, the $N*K*P*Q$ output tensor, and the $N*C*(P+R-1)*(Q+S-1)$ input tensor. Given a set of points in the operation space, Timeloop can determine the set of operands and results for those points by *projecting* the 7D operation points into the 4D dataspace dimensions.

B. Architecture Specification

To describe an architecture in Timeloop, one must specify (a) the hardware organization, i.e., the topology of interconnected compute and storage units, and (b) mapspace constraints that limit the set of mappings allowed by the hardware. We describe the organization specification in this section, and cover mapspace constraints in Section V-D.

Timeloop uses an organization template that is sufficiently parameterizable to model a variety of architectures of interest. The template is a hierarchical tree of storage elements with a set of arithmetic units (such as MACs) at the leaves and a backing store (such as DRAM) holding all of the workload's data at the root. Timeloop supports an arbitrary number of memory levels. For each memory level, the number of instances, entries per instance, bits in each entry, bandwidth, and various other microarchitectural attributes can be specified.

Interconnection network topology is automatically inferred from the storage hierarchy specification; additional micro-architectural properties can be explicitly specified. *Inter-level* networks are either point-to-point or fan-out/fan-in networks capable of *multi-casting* operands and/or *spatially reducing* partial sums (using adder trees) — abilities that have a significant impact on energy efficiency [24]. *Intra-level* networks connect peer-instances within a level and may be used to forward operands and partial sums between neighbors, eliding expensive accesses to a larger parent storage.

Figure 4 shows the organization of Eyeriss [6] specified in the Timeloop format. The configuration uses 256 *PEs* (each with a MAC unit and a private 256-entry register file), a single shared 128KB global buffer, and a backing DRAM. In addition to this organization specification, we also need to model the

```
// === 1D Convolution Workload ===
for r=[0:R):
 for p=[0:P):
  Output[p] += Weight[r] * Input[r+p];

// === Mapping ===
// DRAM
for r3=[0:R3):
 for p3=[0:P3):
  // GBuf
  for r2=[0:R2):
   for p2=[0:P2):
    // Spatial: GBuf->RFile
    parallel_for r1=[0:R1):
     parallel_for p1=[0:P1):
      // RFile
      for r0=[0:R0):
       for p0=[0:P0):
        p = p3*P2*P1*P0 + p2*P1*P0 + p1*P0 + p0;
        r = r3*R2*R1*R0 + r2*R1*R0 + r1*R0 + r0;
        Output[p] += Weight[r] * Input[r+p];
```

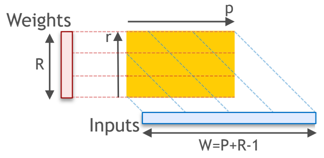


Fig. 5. Example mapping of a 1D convolution onto the Eyeriss organization. Inset: the projections defined by this workload from the operation space to the input and weight dataspaces.

row-stationary dataflow enforced by the architecture. This is done by using *mapspace constraints* described in Section V-D. To understand mapspace constraints, we must first understand the structure of a *mapping* in Timeloop.

C. Mappings

A *mapping* describes the way in which the *operation space* and the associated *dataspaces* are split into *tiles* (chunks) at each level of the memory hierarchy and among multiple memory instances at each level. We use a *loop nest* based mapping representation which can describe all facets of a mapping in a single unified format. Figure 5 shows a 1D convolution workload with input activation of size P and filter size R followed by an example mapping onto the Eyeriss organization from Figure 4. To construct the mapping, the 1D convolution loop nest is split into a number of sections (called *tiling levels*) equal to the the number of storage hierarchy levels, plus the number of levels with spatial fanouts. Each tiling level has a loop corresponding to each dimension in the original workload (though the bound may be 1). The product of all the loop bounds belonging to a dimension must be equal to the final (optionally padded) value of the dimension, e.g., $P = P0*P1*P2*P3$ and $R = R0*R1*R2*R3$.

Using the mapping in Figure 5 as an example, we see:

(1) Loop bounds at each tiling level determine the size of the tile for each dataspace held at that level, e.g., each RFile owns an $R0*P0$-sized operation space tile, which projects onto $R0$, $(P0+R0-1)$, and $P0$-sized weight, input, and output tiles, respectively. Size of the dataspace tiles is constrained by the size of the buffer at each level.

(2) `parallel_for` loops represent spatial partitioning of tiles across instances of a level, e.g., the `parallel_for` loops above distribute the $(R0*R1)*(P0*P1)$ operation space (and the dataspace projections thereof) across $(R1*P1)$ instances of the RFile. This specific mapping results in replication of some input data (called a *halo*) between adjacent RFile instances.

```
constraints = ({
 type = "spatial";         { type = "temporal";
 target = "GBuf->RFile";     target = "RFile";
 factors = "S0 P1 R1 N1";    factors = "R0 S1 Q1";
 permutation = "SC.QK";},    permutation = "RCP";});
```

Fig. 6. Row-stationary dataflow used in Eyeriss [6] described as a set of Timeloop mapspace constraints.

(3) Ordering of loops within a tiling level determines the sequence in which sub-tiles will be delivered from that level to an inner level during execution, e.g., the GBuf owns an $(R0 * R1 * R2) * (P0 * P1 * P2)$-sized operation space tile. Iterating over the $p3$ loop at DRAM results in interesting changes in each GBuff dataspace tile: the weights remain *stationary*, the outputs get replaced each iteration, and the inputs display an overlapping *sliding-window* pattern, which means the DRAM only needs to supply the portion of the tile that is not already present in the GBuf. Inferring these dataspace changes (or *deltas*) is a primary function of Timeloop's access-count model described in Section VI-A.

This representation produces a strictly inclusive tile hierarchy, which may not be optimal. When a dataspace has less reuse at a level, allowing it to *bypass* that level opens up the capacity to other dataspaces, enabling larger tile sizes and potentially resulting in a more optimal mapping. Our mapping specification includes a directive (not shown for brevity) for specifying which dataspaces are allowed to reside at each level.

The approach used for this 1D convolution example can be easily extended to a higher-dimensional space, such as a 7D CNN layer. This unified mapping representation allows us to reason about the space of possible mappings (or *mapspace*) for an architecture in a structured, programmatic manner.

D. Mapspace Constraints

By default, Timeloop assumes that a specified hardware organization is *fully flexible*, i.e., networks and state machines in the hardware are fully programmable, which gives a mapper complete flexibility in partitioning and scheduling arithmetic operations and data movement across the hardware resources. To keep hardware designs simple and efficient, most real architectures are constrained in various ways and only support a subset of the complete mapspace. We model these using mapspace constraints, a generalization of the concept of *dataflows* [6] introduced in prior work. To explain Timeloop's constraints, we augment the Eyeriss architecture specification from Section V-B with a list of constraints in Figure 6. Each constraint targets a single tiling level in the mapping.

Factors in a constraint fix values for loop bounds in the mapping (recall mapping in Figure 5). An un-specified factor gives Timeloop's mapper full flexibility in determining an optimal value. *Permutations* specify loop ordering within a tiling level. Left un-specified (or partially specified), they give Timeloop flexibility to reorder loops.

When applied to a **spatial** tiling level (`parallel_for`), these directives affect the partitioning of the workload space across a hardware spatial dimension. In Figure 6, the `S0 P1 R1 N1` factors disallow parallelism along the P, R and N dimensions and force unrolling along the S (filter height) dimension. The `SC.QK` permutation constraint forces S and C to be unrolled on the physical X-axis of the mesh of hardware PEs, and Q and K on the Y-axis.

When applied to a **temporal** tiling level (`for`), these directives affect tile sizes and data access patterns at that level. In Figure 6, the `R0 S1 Q1` factors force each PE to exhaust the entire R (filter-width) dimension as a temporal loop, and allow it to only map one row of filters and one row of output activations. Additionally, **level bypass** constraints (not shown in the example) dictate whether a level must compulsorily store or bypass a tensor, or whether the mapper is free to explore various alternatives.

Together, the constraints in Figure 6 are sufficient to model a row-stationary dataflow. Once the constraints are set up, Timeloop automatically infers and exploits the various spatio-temporal reuse opportunities described in [6].

E. Mapspace Construction and Search

A mapspace is the set of all legal mappings of a workload onto an architecture. All mappings in a mapspace have the same number of tiling levels, but differ in (a) the values assigned to the loop bounds at each level and (b) the permutation of loops within each level. To construct the mapspace, we must enumerate all possible factorizations of each workload dimension across levels (we call this the *IndexFactorization* sub-space), all possible permutations of loops within a level (the *LoopPermutation* sub-space), and all level bypassing alternatives (the *LevelBypass* sub-space). The Cartesian product of these sub-spaces gives us an *unconstrained* mapspace, which can be quite large due to combinatorial explosion, e.g., for a 7D CNN on a 4-tiling-level architecture the size is $(7!)^4 \times (2^4)^3 \times$ size of the Cartesian product of the co-factor sets for each of the 7 loop bounds. While there are ways to prune this space, e.g., permutations do not matter for the innermost tiling level, and for factors that are 1, the space is still large.

User-specified constraints are accommodated into the mapspace, shrinking the sizes of the underlying sub-spaces. A mapping sampled from the mapspace is therefore guaranteed to obey those constraints. Hardware resource constraints, e.g., whether a set of tensor tiles at a level fit into the size of memory at that level, are checked once a mapping is sampled from the mapspace, and the mapping is rejected if the constraints cannot be met.

A *search* routine samples a mapping from the pruned-and-constrained mapspace, evaluates it using the architecture model (described next in Section VI), and chooses the next mapping to evaluate based on some heuristic. For our experiments, we currently employ either an exhaustive linear search (for small mapspaces) or a random sampling based heuristic (for large mapspaces). More sophisticated search heuristics are planned for future work. For most experiments, we use energy-delay product as the goodness metric, though any of the statistics provided by the model can be trivially used for this purpose.

VI. Timeloop Model

Timeloop's architecture model evaluates a mapping by first analyzing hierarchical *tiles* of data that represent the mapping and measuring the transfer of data between them to obtain *tile-access* counts. Next, these tile-access counts are used to derive the access counts to hardware components, which combined with the energy per access model give performance and energy. The technology model also provides an area estimate based on the architecture specification.

A. Tile Analysis

As explained in Section V-C, loop bounds at each tiling level determine the tile for the operation space, and in turn for each dataspace held at that level. To compute *tile-access* counts, we must determine the volumes of data that must move between the tiles over space and time to execute the schedule dictated by the mapping. To perform this analysis, Timeloop uses a data structure called a *point set* to track tiles of operation and operand/result spaces at each time step and at each instance of a hardware unit (such as a multiplier or a buffer instance).

Consider two consecutive iterations i and $i+1$ at any loop in a mapping (from Figure 5). At each iteration, we can determine the point sets required by the complete sub-nest underneath this loop. We call the set-difference between these point sets a *delta* (see Figure 7). The size of this delta has different connotations for spatial and temporal loops.

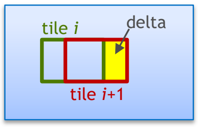


Fig. 7. The delta for tile $i+1$ are points in tile $i+1$ that are not in tile i.

For a temporal (`for`) loop, an empty delta indicates perfect reuse over time (*stationarity*), requiring no additional data movement. A non-empty delta—indicating no reuse or partial reuse (such as in a sliding-window pattern)—represents the incremental data that must be transferred between levels.

For a spatial (`parallel_for`) loop, the deltas represent overlaps between data held at adjacent hardware instances (represented by the iterations i and $i+1$). An empty delta at the same time step indicates a *multicasting* opportunity, while an empty delta at consecutive time steps between adjacent hardware instances indicates a *forwarding* opportunity (such as in a systolic array).

To compute all tile access counts, we need to measure and accumulate deltas over all space (all hardware units) and time (the complete execution of the mapping). A naïve but robust way to do this is to effectively *simulate* the execution of the entire loop nest on the architecture. Sadly, this is unacceptably slow. Fortunately, DNN workloads index into operand and result tensors in a very regular way, and in most cases with each loop index variable showing up only once in the indexing expressions for each tensor. This allows for two optimizations. First, Timeloop only needs to compute the tiles for the first, second and last iterations of each loop. The deltas between other iterations will be of the same shape and size as between the first and second iterations, so the access costs of these other iterations can be algebraically extrapolated. Second, each tile shape is an *axis-aligned hyper-rectangle* within the tensor, which allows for easy delta calculation.

B. Microarchitecture Model

Because of the regular computation and data movement patterns exhibited by DNN layers, buffer accesses and network transfers can usually be statically scheduled, largely avoiding irregular and disruptive bank and network conflicts. Therefore, given a set of tile-access counts, performance can be projected using a throughput/bandwidth based approach, and energy can be estimated using linear transformations of the tile-access counts. The first transformation step is to determine access counts for various microarchitectural structures.

Multiplier accesses are the accesses of the innermost tile of the operation space.

Buffer accesses and *network* ingresses are linear functions of data transfer counts between levels obtained from tile analysis. SRAM buffers can have various aspect ratios and bank configurations, and can be ganged together as vectors for area and energy-efficiency; these configurable micro-architectural parameters are factored into the access count calculations. Partial sums require a read, a write and a *temporal accumulation* operation. Some architectures may elide the first zero-read, and may support local accumulation at some buffers. Network activity includes traffic between levels (including fan-out and multi-cast traffic), as well as traffic between storage instances at the same level (if they forward data to each other). Multi-cast signatures from the tile analysis stage are used to determine the spatial locations of child instances targeted by a specific parent→child data transfer. The architectural topology determines the number of hops required for the data to be routed to the destinations. *Spatial reduction* of output tensors is also derived from network activity.

Address generators are adders surrounded by state machines used by accelerators to generate read and write addresses for storage elements. The number of invocations to these is equal to the number of accesses to the storage element. The bit-width of the adders is log_2 of the number of vector entries in the storage element; this is used by the energy model.

C. Technology-Specific Area/Energy Models

Timeloop supports several user-extensible technology models for area and energy modeling. The nominal TSMC 16nm FinFET-based model used in most of our case studies (except where noted otherwise) has the following components:

(1) A *Memory Model* supports the modeling of memory with different sizes, aspect ratios, number of ports and banks. It also supports different memory implementations, including DRAM, SRAM and register files. The SRAM and register-file component is based on a database of area and energy-per-access costs that is created by generating and measuring a large variety of memory structures with different parameters using an internal memory compiler in TSMC 16nm. The DRAM model is based on average pJ/bit access costs and supports LPDDR4, HBM, DDR4 and GDDR5 technologies.

TABLE I
VALIDATED DNN ACCELERATOR ARCHITECTURES.

	NVDLA-derived [28]	Eyeriss [8]
Dataflow	Weight Stationary	Row Stationary
Reduction	Spatial Reduction	Temporal Reduction
Memory Hierarchy	Distributed and Partitioned Buffer	Centralized L2 Buffer
Interconnect	N/A	Multicast/Unicast
Technology	16 nm	65 nm

(2) An *Arithmetic Model* supports modeling of MACs with different bit-widths. It is also based on a database of area and energy per access costs created by synthesizing and measuring various multiplier and adder designs for different bit-widths in TSMC 16nm. For bit-widths not in the database, Timeloop scales energy per access either quadratically (for multipliers) or linearly (for adders).

(3) A *Wire/Network Model* supports the modeling of on-chip networks with different bit-widths. To model the wire energy consumed for a network hop, area estimates are used to determine the wire distance between the two ends of the hop. This distance is multiplied with a fJ/bit/mm value measured in TSMC 16nm to determine the per-hop energy.

D. Estimating Performance and Energy

Performance is estimated by calculating the number of cycles it would take for each hardware component to complete the workload in isolation. For multipliers, required cycles are equal to the number of MACs in the workload divided by the number of multipliers. For communication interfaces, required cycles are equal to the amount of data flowing through that interface divided by its bandwidth. Buffers, networks and arithmetic units are modeled to be operating in a pipeline. Thus, the overall latency is the maximum of isolated execution cycles across all buffers, networks, and arithmetic units in the hardware. This model, which assumes negligible pipeline stalls, is reasonable for architectures that use double-buffering or more sophisticated techniques like *buffets* [30].

Energy consumption is estimated by multiplying the hardware component access count with the energy per access from the energy model, taking sparsity into account. The total energy consumed by the mapping is the sum of energy consumed by all the components.

E. Extensibility beyond regular workloads/architectures

While our throughput-based performance model and linear energy model are adequate for regular DNN architectures, we believe that our approach of separating a high-level tile analysis from a low-level microarchitecture model allows for extensibility to more complex architectures as well. Tile analysis produces a compact representation of a mappings data access patterns, which can be fed into a non-linear modeling backend if desired, e.g., one with a stochastic model of network conflicts/congestion, or even a full simulation of a network serving those accesses.

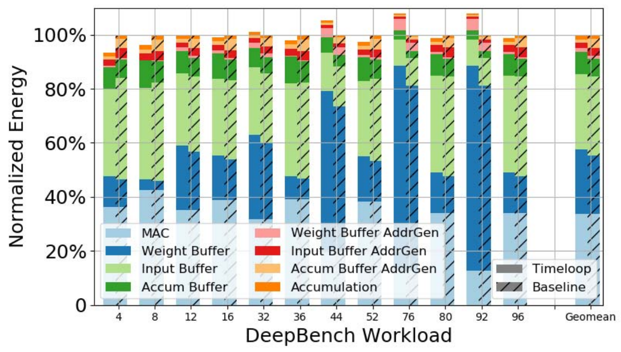


Fig. 8. Energy validation results for NVDLA-derived architecture.

VII. TIMELOOP VALIDATION

We validate Timeloop across a wide range of workloads with two state-of-the-art DNN accelerators for which accurate performance and energy characterizations are available.

A. Architectures

To demonstrate Timeloop's ability to flexibly evaluate diverse DNN architectures, we validate Timeloop against two representative DNN accelerator architectures that are notably different in scale, organization, dataflow and technology. The architectures' key attributes are summarized in Table I.

1) NVDLA-derived Architecture: NVDLA is an open-source hardware accelerator design for automotive applications [28]. To better understand the design trade-offs in DNN accelerators, we designed an in-house, NVDLA-derived architecture in RTL. Similar to NVDLA, it also deploys a weight-stationary dataflow with spatial reduction. However, it features a distributed and partitioned weight and input L1 buffer for better local reuse. We use a detailed in-house simulator designed to accurately model this specific architecture to derive reference performance and energy numbers for validation against Timeloop.

2) Eyeriss Architecture: Eyeriss is one of the first efficient and flexible DNN accelerators [6]. As shown in Table I, it proposes a novel row-stationary dataflow with both a temporal reduction datapath, a centralized L2 buffer, and a flexible network that supports both multicast and unicast. In addition, Eyeriss was designed using a 65nm technology. To validate Timeloop against Eyeriss results, we use the 65nm model from Table IV in [6].

B. Workloads

We validate Timeloop using DNN kernels from the DeepBench benchmark suite [10] augmented with additional synthetic DNN kernels with representative configurations. The DeepBench suite includes 107 DNN workloads capturing computation in convolution, matrix-matrix multiply, and matrix-vector multiply of different dimensions, all of which are commonly used in modern neural networks, e.g., computer vision and speech recognition.

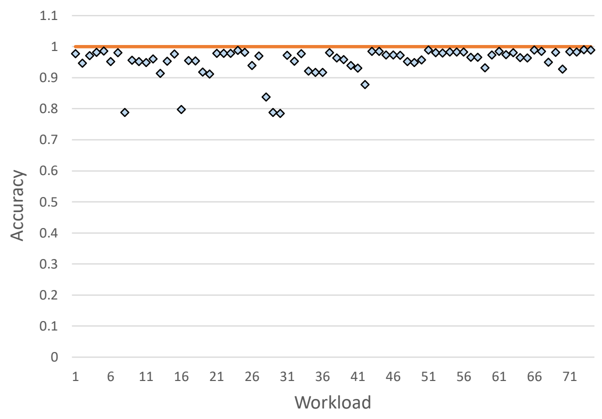


Fig. 9. Performance validation results for NVDLA-derived architecture.

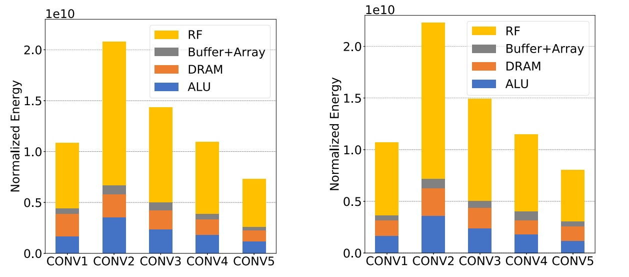
C. Validation Results

Figure 8 shows energy validation results for Timeloop against our NVDLA-derived baseline. The bar on the left shows the energy breakdown from Timeloop, and the bar on the left represents the baseline energy. All the energy is normalized to the total baseline energy. The X-axis represents different workloads from DeepBench. We show a subset of DeepBench workloads in Figure 8 for brevity, though Timeloop accurately captures the energy behavior of the accelerator across the entire benchmark suite, with projections for all 107 workloads within 8% of the baseline.

Figure 9 shows performance validation results for Timeloop against our NVDLA-derived baseline architecture. The X-axis represents a range of synthetic workloads and the Y-axis plots accuracy measured as the cycles reported by Timeloop divided by reference cycles. Accuracy ranges from 78% to 99% with a mean of 95% across all workloads. Recall that Timeloop uses a throughput based performance model that assumes minimal pipeline disruptions from fills and drains. Despite this assumption, accuracy is high (90% to 99%) on all but six outlier workloads, in this case because the hardware uses buffets [30], which provides efficient overlap of compute and data-transfers with minimal additional storage. This can also achieved via double-buffering but at the cost of twice the amount of storage.

The lower accuracy (78% to 88%) for six outlier workloads is a consequence of sub-optimal data layout and transfer orders in the hardware. For example, in one case, the sequence of addresses used by the hardware address-generator filling the Input Buffer was different from the sequence used by the address-generator reading from the buffer, causing unexpected pipeline stalls. We belatedly realized that these are sub-optimal configurations of the hardware, and believe that better configurations would show better correlation with Timeloop's projections.

Figure 10 shows results for the validation experiment against the Eyeriss architecture. Specifically, our objective here is to recreate the experiment and results in Figure 10 of the Eyeriss paper [6], with AlexNet [21] as the workload. The architecture's organization and mapspace constraints were previously described as examples in Sections V-B and V-D respectively. We see that Timeloop's estimation tracks quite closely with the energy reported from the prior study.



(a) Data from [6]

(b) Timeloop Model

Fig. 10. Normalized energy for AlexNet layers on a 256-PE Eyeriss [6] architecture employing a row-stationary dataflow.

VIII. Case Studies

This section demonstrates Timeloop's power and flexibility by evaluating a variety of distinct architectures such as NVDLA [28], Eyeriss [6] and DianNao [5] over a range of applications. We first show how designers can use Timeloop to quickly analyze different application behaviors on a given architecture and quantify its performance and energy efficiency. We then present a sampling of case studies that DNN accelerator designers can conduct with Timeloop, e.g., understanding the impact of technology scaling on the mapping strategy, optimizing the energy efficiency through tailoring the memory hierarchy for the dataflow, and systematically comparing and quantifying the differences between existing DNN accelerator architectures.

A. Case Study: Workload Characterization

We first demonstrate how DNN accelerator designers can use Timeloop to characterize an architecture over a suite of workloads and derive a host of insightful statistics. Specifically, we use Timeloop to evaluate all DeepBench workloads running on the NVDLA architecture. Figure 11 shows a detailed characterization of each workload with its optimal mapping found using Timeloop. The Y-axis on the left shows the normalized total energy over the MAC energy, while the one on the right shows the *algorithmic reuse* of each benchmark, defined as the number of MACs divided by the minimum number of DRAM accesses, i.e., the total size of inputs, weights and outputs. We also show the MAC utilization on the top where we achieve close-to-1 utilization in most of cases except the ones with shallow input ($C < 64$) and output ($K < 16$) channels as NVDLA spatially maps C and K to its MACs. We observe that energy is dominated by DRAM for workloads with low reuse. On the contrary, workloads with high reuse are more affected by the efficiency of on-chip components. This study shows how Timeloop allows DNN accelerator designers to quickly evaluate a DNN accelerator across a wide range of application without having to write RTL or even a C++ model.

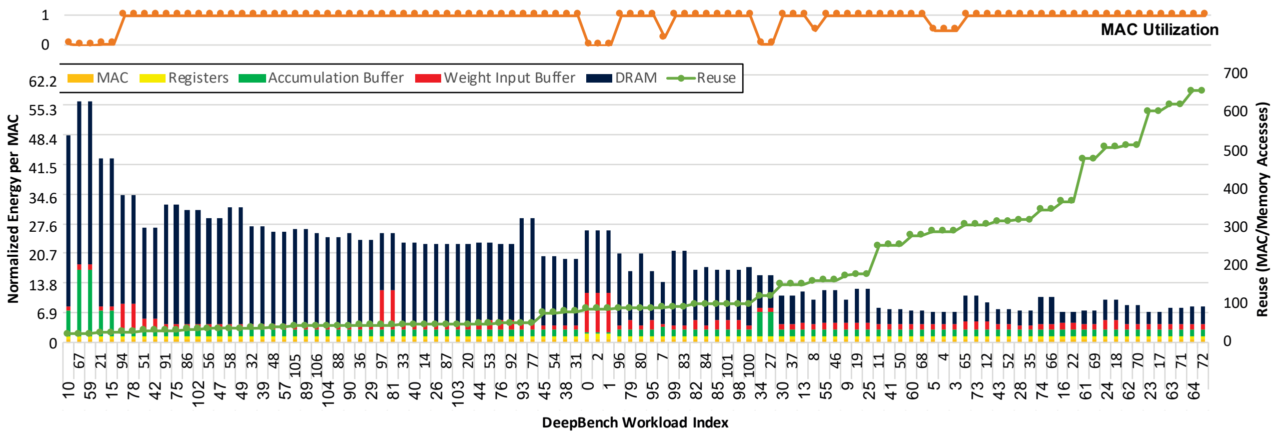


Fig. 11. Energy/MAC breakdown for DeepBench workloads sorted by algorithmic reuse.

B. Case Study: Impact of Technology

Technology differences change the relative energy and area costs of different components, e.g., arithmetic and memory, which may in turn change the optimal mapping of a workload on an architecture. This study evaluates Timeloop's Eyeriss configuration with different technology models and highlights the importance of a mapper in the evaluation of DNN architectures. Specifically, we compare energy efficiency and different mapping choices using both Eyeriss' original 65nm energy model and our 16nm energy model.

Figure 12(a) shows the energy breakdown of Eyeriss architecture running AlexNet layers with 65nm and 16nm models. Energy is normalized to the total energy of each layer. In both configurations, we use the same optimal mapping from 65nm (65map). We see that technology change causes a re-distribution of energy between hardware components. In addition, we see that the optimality of mappings does *not* trivially carry over across different technologies—the optimal mapping for 65nm is sub-optimal for the 16nm model, as illustrated in Figure 12(b). We can reduce the overall energy up to 22% by re-evaluating the mapspace with the 16nm model and finding the optimal mapping for that technology (16map).

C. Case Study: Memory Hierarchy Optimization

The Eyeriss architecture we have modeled so far uses a 256-entry register file (RF) in each PE as the innermost storage level, which is shared across all dataspaces and contributes a significant amount of energy consumption, as shown in Figure 12. In this study, we explore two designs that aim to reduce the RF energy consumption by (1) adding an additional one-entry register at the innermost storage level, and (2) partitioning the shared RF into separate ones for inputs, weights and partial sums. As the row-stationary dataflow has high temporal locality in the RF for inputs and partial sums, the RFs for inputs and partial sums are allocated with 12 and 16 entries, respectively, to take advantage of a small capacity to reduce their access cost. The rest of the entries are allocated for the weight. This is inspired by how Eyeriss is actually implemented in [8], which is slightly different from the model in [6].

Figure 13 shows the normalized energy for various workloads (with batch size of 1) of the three Eyeriss variants. For each layer, the bars from left to right are Eyeriss with: (1) a shared RF, (2) a shared RF and an additional register, and (3) a partitioned RF. The result shows that the memory

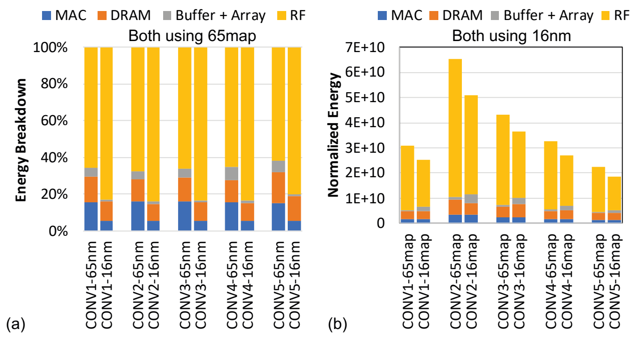


Fig. 12. (a) Energy breakdown of the same *mapping* with different technologies, i.e., 65nm and 16nm. (b) Normalized energy of the same *technology* 16nm with different mapping choices. *65map* is the optimal mapping with 65nm model, while *16map* is the optimal mapping with 16nm model.

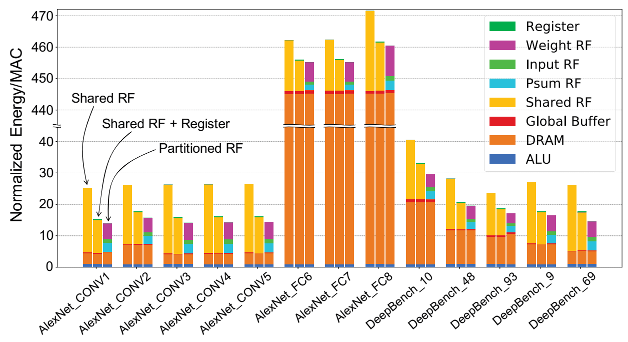
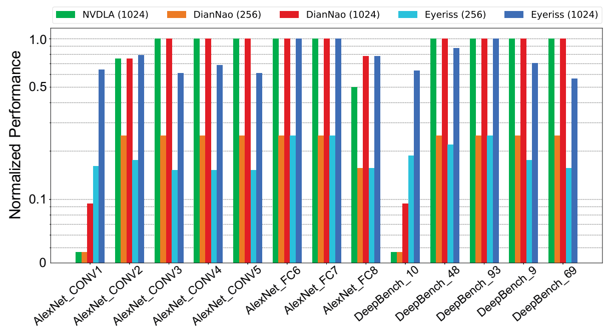
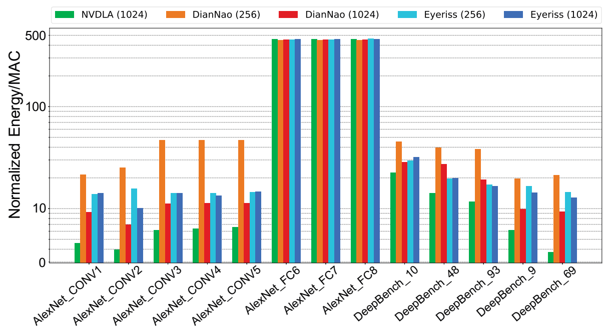


Fig. 13. Normalized energy per MAC for three different Eyeriss designs. For each layer, the three bars from left to right are Eyeriss with (1) a shared RF, (2) a shared RF and an additional register, and (3) a partitioned RF.



(a) Normalized Performance



(b) Normalized Energy/MAC

Fig. 14. Performance and Energy Efficiency Comparisons of NVDLA, DianNao, and Eyeriss.

hierarchy optimizations in (2) and (3) reduce the total energy consumption in all the workloads. The improvement is more pronounced in convolutional layers, which show an over 40% reduction in energy. This case study shows that the co-design of dataflow and memory hierarchy implementation is crucial to the energy efficiency of an architecture; Timeloop can effectively guide this design space exploration.

D. Case Study: Modeling of Existing Architectures

In this study, we use Timeloop to model and compare the performance and energy efficiency of three representative architectures from prior work: NVDLA, DianNao, and Eyeriss. Since NVDLA has more PEs (1024 PEs) than the nominal configurations of DianNao and Eyeriss (256 PEs), we also model *scaled* variants of DianNao and Eyeriss with 1024 PEs each. Increasing the number of PEs scales the multipliers, buffers and network, but results in each architecture occupying a different silicon area from NVDLA (because of organizational differences). Therefore, we then adjust the buffer sizes to align the final area with NVDLA. Figure 14 shows the normalized performance and energy of the three architectures.

First, NVDLA demonstrates higher performance and energy efficiency compared to the other two architectures except for AlexNet CONV1 and DeepBench workload 10, both of which have shallow input channels (C). As both NVDLA and DianNao map input channels spatially across PEs, when the network has smaller number of input channels, not all the PEs can be utilized during its execution. In contrast, Eyeriss shows a consistent performance across different workloads thanks to its flexible mapping scheme. Overall, this demonstrates that there is no single architecture in this experiment that is universally better than the others.

In addition, the scaled version of DianNao demonstrates a higher energy efficiency and better performance across all the workloads than its default version. This is mainly due to the additional spatial reuse of inputs and the efficient spatial reduction of partial sums in the larger PE array, which greatly amortizes the cost of accessing the on-chip buffers.

On the contrary, while the performance of Eyeriss improves with more PEs, its energy efficiency stays relatively the same between the scaled and default configurations. This is due to the fact that most of the energy in Eyeriss is consumed by the RF in each PE instead of shared buffers. As a result, its energy scales approximately with the number of PEs.

The case studies in this section demonstrate Timeloop's capability for modeling and fairly comparing a wide range of architectures on a diverse set of workloads. The results of such studies should be useful to inform the design of future DNN accelerator architectures.

IX. Conclusions and Future Work

We presented Timeloop, an infrastructure for modeling and evaluating a rich design space of DNN accelerator architectures. We demonstrated how to describe DNN architectures using a generic template, characterize workloads such as convolutional layers, comprehensively describe mappings using a loop-nest-style representation, and efficiently enumerate the space of feasible mappings by combining workload and architectural constraints. We also described computationally inexpensive and modular techniques for modeling the data transfers and arithmetic for a given mapping, which gives energy and performance estimates with enough speed and accuracy to rapidly explore architectural trade-offs, while including the important step of using an appropriate mapping for each accelerator design. Overall, this provides a practical and powerful automated tool for understanding the complex trade-offs in DNN accelerator design across a wide range of workloads and helps navigate those trade-offs in a systematic fashion.

Future work includes modeling inter-layer relationships to find globally-optimal solutions for full networks (as in [2]). Also, although Timeloop already accounts for the energy savings due to sparsity, future work includes modeling architectures that save both time and energy [1], [15], [29], [39]. Other future work includes modeling accelerators for other domains like graph analytics [40] and sparse tensor algebra [20].

The generality of Timeloop opens the door to a variety of research directions in accelerator architecture design, especially for accelerators with mapping dependent behavior. We are in the process of releasing the infrastructure to the community under an open-source license.

Acknowledgments

We thank Michael Pellauer and Christopher W. Fletcher for sharing their insights on dataflows over numerous discussions. We also appreciated the valuable insights William J. Dally provided. Finally, we thank the anonymous reviewers for their helpful and insightful comments.

Distribution Statement A. This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-Neuron-Free Deep Convolutional Neural Network Computing. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 1–13, June 2016.

[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer CNN accelerators. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2016.

[3] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep Speech 2: End-To-End Speech Recognition in English and Mandarin. https://arxiv.org/abs/1512.02595, 2015.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association.

[5] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pages 269–284, March 2014.

[6] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 367–379, June 2016.

[7] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators. *IEEE Micro's Top Picks from the Computer Architecture Conferences*, 37(3), May-June 2017.

[8] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. Eyeriss: An Energy-efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *Proceedings of the International Solid State Circuits Conference (ISSCC)*, February 2016.

[9] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural Language Processing (Almost) From Scratch. In *arxiv.org*, 2011.

[10] DeepBench. https://github.com/baidu-research/DeepBench.

[11] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Hannun, and S. Satheesh. Persistent RNNs: Stashing Recurrent Weights On-Chip. In *ICML*, June 2016.

[12] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 92–104, June 2015.

[13] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2017.

[14] A. Graves and J. Schmidhuber. Framewise Phoneme Classification With Bidirectional LSTM and Other Neural Network Architectures. In *Neural Networks*, 2005.

[15] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark Horowitz, and Bill Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 243–254, June 2016.

[16] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng. Deep Speech: Scaling Up End-To-End Speech Recognition. https://arxiv.org/abs/1412.5567, 2014.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[18] 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. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[19] Liu Ke, Xin He, and Xuan Zhang. Nnest: Early-stage design space exploration tool for neural network inference accelerators. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '18, pages 4:1–4:6, New York, NY, USA, 2018. ACM.

[20] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. The Tensor Algebra Compiler. In *Proc. OOPSLA*, 2017.

[21] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*, December 2012.

[22] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 461–475, New York, NY, USA, 2018. ACM.

[23] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 521:436–444, May 2015.

[24] C. Lee, Y.S. Shao, J-F Zhang, A. Parashar, J. Emer, S.W. Keckler, and Z. Zhang. Stitch-x: An accelerator architecture for exploiting unstructured sparsity in deep neural networks. In *SysML Conference*, 2018.

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

[26] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[27] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of FPGA-based deep convolutional

neural networks. In *21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016.

[28] Nvidia. NVDLA Open Source Project, 2017.

[29] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[30] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Architectural Support for Programming Languages and Operating Systems (to appear)*, ASPLOS '19, 2019.

[31] Benoît Pradelle, Benoît Meister, M. Baskaran, Jonathan Springer, and Richard Lethin. Polyhedral Optimization of TensorFlow Computation Graphs. In *Workshop on Extreme-scale Programming Tools (ESPT)*, November 2017.

[32] Atul Rahman, Sangyun Oh, Jongeun Lee, and Kiyoung Choi. Design space exploration of FPGA accelerators for convolutional neural networks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[33] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Saekyu Lee, Jose Miguel Hernandez Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling Low-Power, High-Accuracy Deep Neural Network Accelerators. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 267–278, June 2016.

[34] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to FPGAs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2016.

[35] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing CNN accelerator efficiency through resource partitioning. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[36] Kodai Ueyoshi, Kota Ando, Kentaro Orimo, Masayuki Ikebe, Tetsuya Asai, and Masato Motomura. Exploring optimized accelerator design for binarized convolutional neural networks. In *International Joint Conference on Neural Networks (IJCNN)*, 2017.

[37] Kaiyi Yang, Shihao Wang, Jianbin Zhou, and Takeshi Yoshimura. Energy-efficient scheduling method with cross-loop model for resource-limited cnn accelerator designs. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017.

[38] Tien-Ju Yang, Yu-Hsin Chen, Joel Emer, and Vivienne Sze. A method to estimate the energy consumption of deep neural networks. In *Asilomar Conference on Signals, Systems and Computers*, 2017.

[39] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-X: An Accelerator for Sparse Neural Networks. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, October 2016.

[40] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. GraphIt-A High-Performance DSL for Graph Analytics. *arXiv preprint arXiv:1805.00923*, 2018.