Understanding Reuse, Performance, and Hardware Cost of DNN Dataflows: A Data-Centric Approach

Hyoukjun Kwon
Georgia Institute of Technology
Atlanta, Georgia
hyoukjun@gatech.edu

Prasanth Chatarasi
Georgia Institute of Technology
Atlanta, Georgia
cprasanth@gatech.edu

Michael Pellauer
NVIDIA
Westford, Massachusetts
mpellauer@nvidia.com

Angshuman Parashar
NVIDIA
Westford, Massachusetts
aparashar@nvidia.com

Vivek Sarkar
Georgia Institute of Technology
Atlanta, Georgia
vsarkar@gatech.edu

Tushar Krishna
Georgia Institute of Technology
Atlanta, Georgia
tushar@ece.gatech.edu

ABSTRACT

The data partitioning and scheduling strategies used by DNN accelerators to leverage reuse and perform staging are known as *dataflow*, which directly impacts the performance and energy efficiency of DNN accelerators. An accelerator microarchitecture dictates the dataflow(s) that can be employed to execute layers in a DNN. Selecting a dataflow for a layer can have a large impact on utilization and energy efficiency, but there is a lack of understanding on the choices and consequences of dataflows, and of tools and methodologies to help architects explore the co-optimization design space.

In this work, we first introduce a set of data-centric directives to concisely specify the DNN dataflow space in a compiler-friendly form. We then show how these directives can be analyzed to infer various forms of reuse and to exploit them using hardware capabilities. We codify this analysis into an analytical cost model, MAESTRO (Modeling Accelerator Efficiency via Spatio-Temporal Reuse and Occupancy), that estimates various cost-benefit tradeoffs of a dataflow including execution time and energy efficiency for a DNN model and hardware configuration. We demonstrate the use of MAESTRO to drive a hardware design space exploration experiment, which searches across 480M designs to identify 2.5M valid designs at an average rate of 0.17M designs per second, including Pareto-optimal throughput- and energy-optimized design points.

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**; • **Hardware** → *Modeling and parameter extraction*.

KEYWORDS

Neural networks, Dataflow, Cost modeling

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358252

ACM Reference Format:
Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflows: A Data-Centric Approach. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3352460.3358252

1 INTRODUCTION

Deep neural networks (DNNs) are being deployed at an increasing scale—across the cloud and IoT platforms—to solve complex regression and classification problems in image recognition [41], speech recognition [5], language translation [46], and many more fields, with accuracy close to and even surpassing that of humans [16, 20, 44]. Tight latency, throughput, and energy constraints when running DNNs have led to a meteoric increase in hardware accelerators.

DNN accelerators achieve high performance by exploiting parallelism over hundreds of processing elements (PEs) and high energy efficiency by maximizing data reuse within PEs and on-chip scratchpads [1, 9, 11, 19, 31, 38]. For a specific DNN workload and a hardware accelerator, the achieved utilization and data-reuse directly depends on (1) how we schedule the DNN computations (e.g., choice of loop transformations) and (2) how we map computations across PEs. These two components are collectively referred to as *dataflow* in the accelerator literature [11, 24, 25, 31]. It has been shown that the energy cost of moving data exceeds the cost of computation [11, 17], and so understanding and optimizing dataflow is a critical component of DNN accelerator design, as it directly determines how data is transferred between multipliers (L0), staged in local buffers (L1), and in the global buffer hierarchy (L2 and beyond).

The performance and energy efficiency of DNN accelerators depend on (1) target DNN model and its layers types/dimensions, (2) dataflow, and (3) available hardware resources and their connectivity. These three dimensions are tightly coupled, and optimizing DNN accelerators across these dimensions is a challenging task. For example, a dataflow that exploits input channel parallelism [1] in convolutional neural networks (CNNs) may not achieve high utilization on layers with a small number of channels. Alternatively, dataflows that require more transfer bandwidth than the network-on-chip (NoC) provides may result in under-utilization of the hardware. In such cases, increasing the L1 scratchpad size may allow the same dataflow to require less data bandwidth, but this larger L1

may increase area and energy consumption. Thus, co-optimizing the hardware microarchitecture and the dataflows it supports is one of the primary optimization targets for any accelerator design. This remains an open challenge, as observed by the number of novel dataflows and microarchitectures that continue to be proposed recently [12, 17, 25, 27].

Regrettably, these proposals do not cover the complete space of dataflows at an exhaustive-enough level to serve as a reference for architects designing custom accelerators within a variety of constraints. In contrast, recent proposals on compilation [10, 33] and analysis tools [30] for DNNs analyze a broad space of software mappings of a DNN workload onto a given architecture, but the relationship between software mappings and hardware dataflows is not elucidated, and these black-box tools do not provide architects with intellectual intuitions on the consequences of dataflow selection and their impact on reuse. In fact, the very term "dataflow" is used in an inconsistent manner across both architecture and analysis proposals. Architects are thus left with an incomplete and unstructured set of intuitions on dataflows and the complex interplay between dataflow and microarchitecture choices.

In this paper, we seek to remedy this situation by providing a thorough set of insights on the choices and consequences of dataflow selection and their interplay with microarchitectural alternatives, and a structured mechanism to reason about them quantitatively. To that end, we make the following specific contributions.

First, we introduce a *data-centric* notation to represent various accelerator dataflows with data mappings and reuses being first-class entities, unlike the compute-centric notation used by prior proposals which infer the data reuses from a loop-nest representation [12, 25, 26, 30]. These data-centric directives can express a wide range of data-reuses (across space, time, and space-time) over arbitrary hierarchies of PEs for both dense and sparse DNN layers such as convolutions, LSTMs, and fully-connected layers. We believe that our data-centric notation can complement the commonly used loop-nest notation, i.e., our notation can be viewed as an intermediate representation (IR) which can be extracted from a high-level loop-nest notation or specified directly.

Second, we show how these data-centric directives can be used to reason about reuse in a structured manner. We demonstrate the relationship between each directive, the specific form of algorithmic reuse exposed by the directive, and the potential ways to exploit that reuse using a hardware capability to improve efficiency. This analysis covers the complete space of ways in which any dataflow can exploit reuse.

Third, we introduce an analytical cost model named MAESTRO (Modeling Accelerator Efficiency via Spatio-Temporal Reuse and Occupancy) that programmatically implements the above analysis. MAESTRO takes as input 1) a DNN model with a set of layers, 2) a dataflow description for each layer specified using our proposed directives, and 3) the hardware configuration. Based on these inputs, MAESTRO outputs estimates of end-to-end execution time, energy (including all compute, buffer, and interconnect activities), NoC costs, and so on. A key challenge in our proposed approach is to provide a cost estimation that is both efficient and sufficiently precise to effectively support design space exploration. We demonstrate MAESTRO's abstract hardware model and analytic model to be within 90-95% accuracy of actual open-source RTL [24] while being

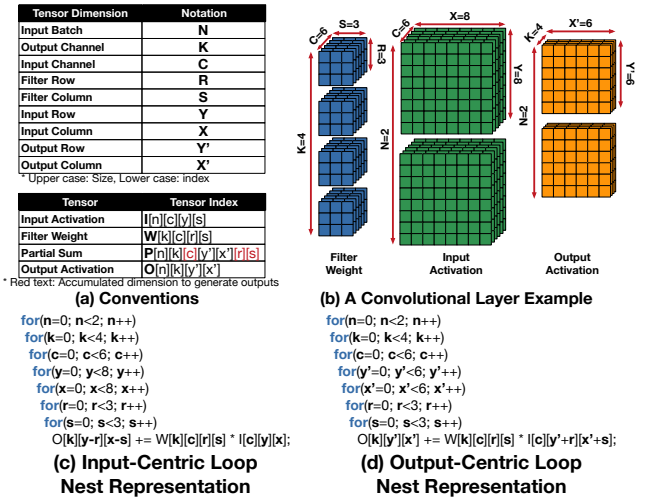


Figure 1: Convolutional layer example

1029-4116× faster (10ms to run MAESTRO versus 7.2-28.8 hours for an equivalent RTL simulation on a workstation with Xeon E5-2699 processor and 64GB memory).

Finally, we demonstrate how the MAESTRO cost model can be used by accelerator designers to determine Pareto-optimal parameters for an accelerator with a given area, energy, or throughput budget. For a NVDLA [1]-like dataflow (KC-Partitioned in Table 3) in VGG16 [42] CONV layer 11, we see up to a 2.16× difference in power consumption between energy- versus throughput-optimized design points. The energy-optimized design employs 10.6× more SRAM and 80% the PEs of the throughput-optimized design. This leads to an energy-delay product improvement of 65%, with 62% throughput. The range of these numbers is a concrete example of the significance of this problem for accelerator architects.

2 BACKGROUND

To understand the cost-benefit tradeoffs of various approaches to compute convolutions, we discuss core concepts related to data reuse and dataflows in the context of DNN accelerators.

2.1 Tensors in DNNs

We present an example of a multi-channel 2D convolution in Figure 1 that involves seven data dimensions across three data structures: input/output activation and weight tensors. Although our approach can be applied to various DNN layers—CONV2D, fully-connected (FC), LSTM, separable convolution, and so on—we focus on CONV2D and its variants in this paper because convolutional neural networks (CNNs) are popular, and CONV2D accounts for more than 90% of overall computation in CNNs [11, 14].

Tensors in DNNs are addressed using seven dimensions in a complex manner. For example, the row/column indices of output can be deduced using input row/column and filter row/column indices (i.e., an input-centric view of the convolution loop nest). Also, the input channel index c appears in both filter and input activation, and the output channel k appears in both filter and output activation. We call these dimensions *coupled* to these indices, as the position in the data space changes when the index is modified. Because of these specific data access patterns, we can transform the loop nest to keep

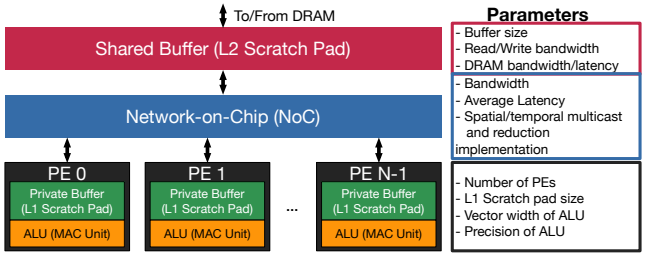


Figure 2: Abstract DNN accelerator architecture model which is pervasive in many state-of-the-art accelerators [3, 11, 19, 31, 38]. The illustrated base architecture can be hierarchically organized.

one of the data structures *stationary* over a range of space or time (i.e., unchanged in a local buffer), which can significantly reduce global/local buffer access counts in DNN accelerators, as well as energy consumption by keeping local wires unchanging.

2.2 DNN Accelerators

DNN accelerators are specialized architectures to run DNN applications with high throughput and energy efficiency. As described in Figure 2, most DNN accelerators employ hundreds of processing elements (PEs) to exploit inherent parallelism in DNN applications. PEs typically include scratchpad memories (L1) and ALUs that perform multiply-accumulate operations (MACs). To reduce energy- and time-consuming DRAM accesses, most DNN accelerators also include a shared scratchpad buffer (L2) large enough to stage data to feed all the PEs. Shared L2 buffer and PEs are interconnected with a network-on-chip (NoC). Our approach supports a wide range of interconnect designs in the NoC module. For example, a systolic array could be represented as a 2D array that provides unidirectional links toward East and South. Depending on the hardware parameters selected, our approach can support architecture designs that can efficiently execute a wide range of DNN operations, including convolutions, because it enables exploiting not only parallelism but also data reuse via buffers and forwarding/multicasting NoCs.

2.3 Data Reuse Taxonomy

We observe that data reuse originates from two behaviors of DNN accelerators over time and space - multicasting (input tensors) and reduction (output tensors).

Multicasting. Spatial multicasting reads a data point from a buffer only once, spatially replicates the data point via wires, and delivers the data point to multiple spatial destinations (i.e., PEs), which reduces expensive remote buffer accesses and saves energy. Likewise, temporal multicasting also reads a data point from a large remote buffer only once, temporally replicates the data point via a smaller local buffer, and delivers the data point to multiple temporal destinations (i.e., different time instances) at the same PE, which also reduces expensive remote buffer accesses and saves energy.

Reduction. Spatial reduction accumulates partial outputs from multiple spatial sources and spatially accumulates them via multiple compute units (e.g., an adder tree or reduce-and-forward). Similarly, temporal reduction accumulates partial outputs from multiple temporal sources (i.e., partial sums computed at different time) and temporally accumulates them via an accumulation register or buffer (e.g., accumulation buffer in TPU [19]).

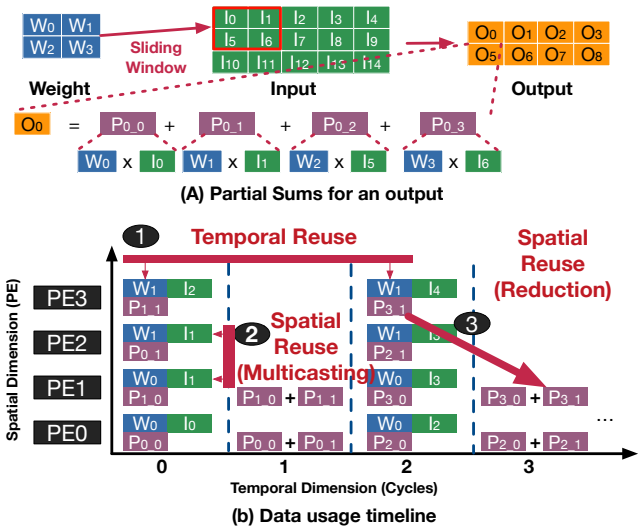


Figure 3: An operational example of a weight-stationary style accelerator with four PEs. For simplicity, input/output channels and batch are omitted. A 2x2 kernel (R=2, S=2) is used in this example.

2.4 Dataflow Definition and Example

In order to leverage these opportunities, the accelerator must schedule operations such that the PEs proceed through the data tensors in a coordinated fashion, which can be viewed as transformations (e.g., ordering and tiling) applied to the convolution in Figure 1, along with a partitioning of data to PEs. Such schedules are termed as *dataflows* in prior work [11], which categorizes dataflows into classes based on the tensor which is scheduled to change least frequently, e.g., weight-stationary, output-stationary, and input-stationary.

Figure 3 shows an example weight-stationary dataflow run on four PEs. We can observe that W_1 is multicast across time (temporal multicasting), I_1 is multicast across PEs (spatial multicasting), and P_{3_1} is reduced across space and time. That is, the example accelerator temporally reuses W_1 and spatially reuses I_1 and P_{3_1}. Note that the name "weight-stationary" conveys intuition and a high-level characterization of scheduling strategy, but detailed insight and analysis requires more precise description.

Chen et al. [12] refine the definition of dataflow by additionally specifying that two schedules which differ only in the concrete bounds should be considered *instances* or *mappings* of the same dataflow. This is an important distinction, as it allows families of accelerators to be categorized together even if they have different buffer sizes—i.e., a mobile chip and a datacenter chip may use the same traversal orderings despite large differences in tile size. For brevity, for the remainder of this work, we make no distinction between schedules with fully-specified or partially unspecified concrete bounds but refer to them all as dataflows.

2.5 Existing Expressions of Dataflow

To convey the scheduling decisions of a particular architecture, dataflows have been expressed as *loop nests*, a syntax that resembles a simple imperative programming language with explicit parallelism, as presented in Eyeriss v2 [12]. We term the loop nest notation a *compute-centric* representation since the data movement is implicit from the loop order and the explicit parallelism specified by the

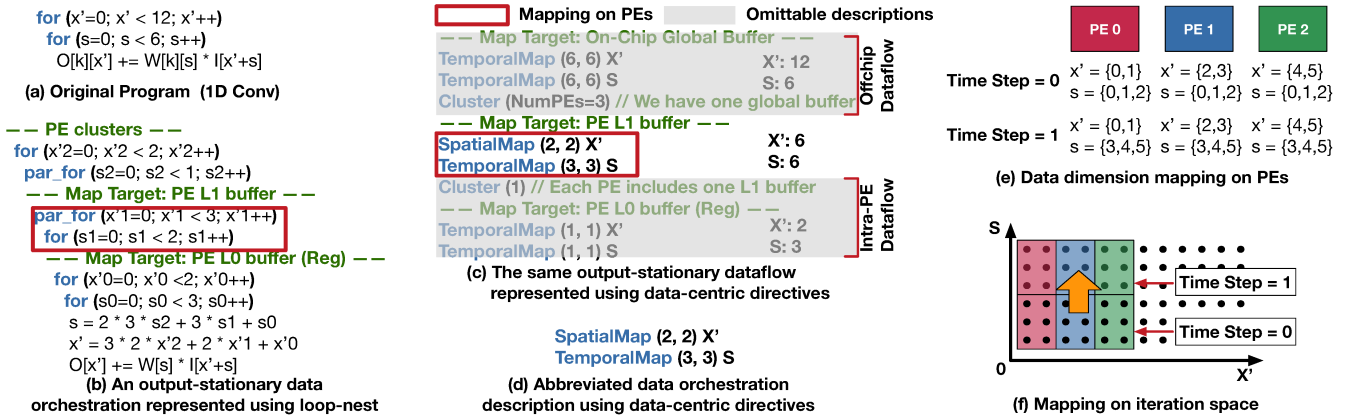


Figure 4: An example 1D convolution and an example output-stationary dataflow on the convolution. We represent the dataflow (b) in loop nest and (c) data-centric directives. In (c), gray boxes represent omittable descriptions, which can be inferred (upper gray box) or do not affect the data reuse over PEs (lower gray box). (d) shows an abbreviated form of the dataflow description in data-centric directives. (e) and (f) show resulting mapping on PEs and iteration space, whose dots correspond to computation (or, partial sums).

programmer. The loop order dictates the schedule (or, ordering) of computations, the explicit annotation of loops with `parallel-for` captures parallelism, and the combination of loop ordering, tiling, and parallelism enables data reuse. Therefore, architects started to explore optimized loop nests encompassing all of the three aspects; loop order, parallelism, and tiling. For example, Eyeriss v2 [12] describes its dataflow in a 22-dimensional loop nest.

Compute-centric representation including the polyhedral model has been a huge help to compilers in estimating reuses in guiding optimal loop transformations for both parallelism and locality [8, 32, 36, 37, 39, 40, 45]. Those works provide sufficiently accurate cost estimations to drive a series loop transformation in a compiler. However, they do not precisely model data reuse, so therefore computing throughput and energy-efficiency with high accuracy is challenging for those works. Bao et al. [7] developed an analytical model to accurately estimate cache behavior (thereby computing reuses) for a class of affine programs that can be precisely analyzed by a polyhedral model at compile time. However, they use heavyweight linear-algebra frameworks within the polyhedral model to compute reuse, thereby making it impractical to use these techniques on real large applications. Also, it is very challenging for the polyhedral-based frameworks to compute reuse arising from array subscripts involving non-affine expressions or complex subscripts, such as modulus operations which are common in strided convolutions.

In addition, although there exists a body of past compiler work that performs reuse analysis on sequential programs [7, 8, 32, 36, 37, 39, 40, 45], they lack the ability to analyze loop nests with explicit parallelism, while DNN dataflows often contain multiple levels of parallelism. Also, those past works did not consider spatial reuse (which does not refer to the spatial locality in cache-based architectures but data reuse via wires or across PEs) that leverages multicasting and reduction support of accelerators, which plays a key role in estimating the overall throughput and energy efficiency of spatial DNN accelerators.

Such limitations and challenges motivate us to explore an alternative intermediate representation (IR) of dataflows, a *data-centric* representation where data movement and organization are first-class entities. Since data movement is explicit in the data-centric representation, our analytical model becomes simpler and relatively faster as there is no need to leverage heavyweight linear-algebra frameworks to precisely estimate data movement/reuse behavior.

3 DESCRIBING DATAFLOWS

Our data-centric representation consists of four key directives – 1) spatial map, 2) temporal map, 3) data movement order, and 4) clusters. We briefly explain all the key directives using 1D convolution (shown in Figure 4 (a)) as a pedagogical example, and then discuss various hardware implementation choices for supporting a wide range of data-reuse across space, time, and space-time.

3.1 Data-Centric Representation

We define the dataflow of an accelerator design to consist of two major aspects – (1) the schedule of DNN computations (e.g., choice of loop transformations) across time for exploiting a wide range of reuse, and (2) the mapping of the DNN computations across PEs for parallelism. The representation is based on four key components, and we briefly discuss the first three components below. The fourth component, `Cluster`, will be introduced in Section 3.2.

(1) **Spatial Map(size, offset) α** specifies a distribution of dimension α (e.g., `R`, `X`) of a data structure across PEs, where `size` refers to the number of indices mapped in the dimension α to each PE, and `offset` describes the shift in the starting indices of α across consecutive PEs.

(2) **Temporal Map(size, offset) α** specifies a distribution of dimension α of a data structure across time steps in a PE, and also the mapped chunk of dimension indices is the same across PEs in a time step. The `size` refers to the number of indices mapped in the dimension α to each PE, and `offset` describes the shift in the starting indices of α across consecutive time steps in a PE.

(3) **Data Movement Order:** The sequence of spatial and temporal maps in the dataflow specification dictate the order of

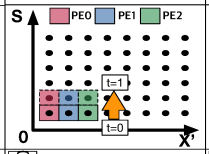
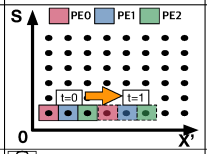
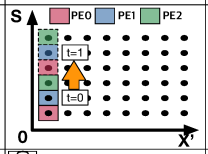
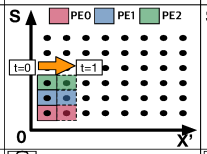
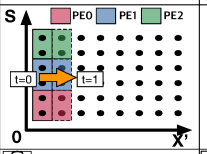
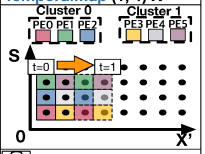
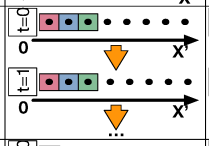
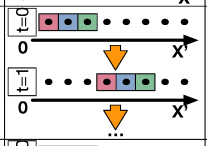
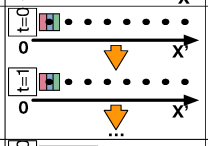
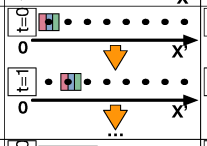
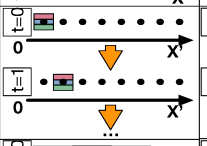
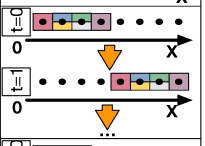
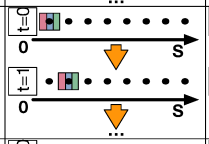
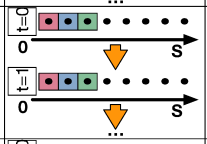
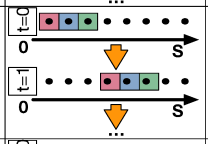
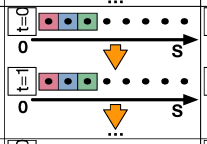
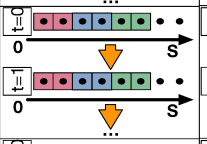
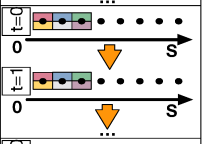
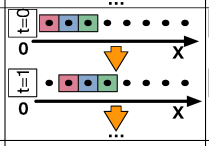
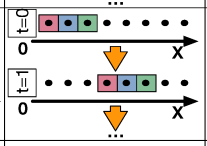
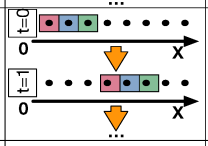
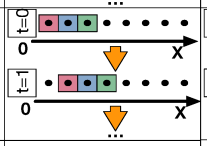
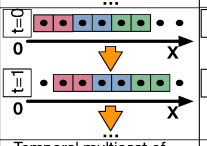
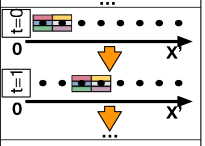
Dataflow ID	A	B	C	D	E	F
Mapping	SpatialMap (1, 1) X' TemporalMap (1, 1) S	TemporalMap (1, 1) S SpatialMap (1, 1) X'	TemporalMap (1, 1) X' SpatialMap (1, 1) S	SpatialMap (1, 1) S TemporalMap (1, 1) X'	SpatialMap (2, 2) S TemporalMap (1, 1) X'	TemporalMap (3, 3) S SpatialMap (1, 1) X' Cluster (3) SpatialMap (1, 1) S TemporalMap (1, 1) X'
Iteration Space (Partial sums)						
Output Featuremap Data Space						
Filter Weight Data Space						
Input Featuremap Data Space						
Temporal Reuse	- Temporal reduction of outputs (Output stationary)	- Temporal multicast of weights (Weight stationary)	- Temporal reduction of outputs (Output stationary)	- Temporal multicast of weights (Weight stationary)	- Temporal multicast of weights (Weight stationary) - Partial temporal multicast of inputs (e.g., X =3 is used by PE1 over t=0 and 1)	- Temporal multicast of weights (Weight stationary)
Spatial Reuse	- Spatial multicast of filter weights	- Spatial multicast of filter weights	- Spatial reduction of outputs	- Spatial reduction of outputs	- Spatial reduction of outputs	- Spatial reduction of outputs
Informal Dataflow Name	Output-Stationary	Weight Stationary	Collaborative Output-Stationary	Collaborative Weight-Stationary	Tiled Collaborative Weight-Stationary	Clustered Tiled Collaborative Weight-Stationary

Figure 5: The impact of directive order, spatial/temporal maps, tile sizes, and clustering over 1D convolution presented in Figure 4. The first row shows mapping described using our data-centric directives. The second row shows iteration spaces whose points correspond to each partial sum. In row three to five, we show data mapping of each data structure. Finally, we describe temporal and spatial reuse opportunities from each mapping.

data movement, i.e., the change of the data mappings to PEs across time.

We demonstrate reuse opportunities presented by various dataflows using the 1D convolution example in Figure 4(a). We start by creating a unique dataflow for this program by the loop nest representation in Figure 4(b), assuming the accelerator has 2-level hierarchy (L0 register at PE + L1 local scratchpad buffer). The two loops enclosed in the red box are indicative of the mapping over the PEs, and their corresponding data-centric representation is in Figure 4(c) and (d).

As can be seen from Figure 4(e), the data elements corresponding to outputs (dimension `X'`) is spatially distributed across three PEs, i.e., each PE receives different chunks of two output elements. This particular data distribution can be captured with our spatial map directive with size and offset parameters being 2, resulting in `SpatialMap(2,2) X'` where `X'` is the first dimension of output data structure. Also, the data elements corresponding to weights (dimension `S`) is replicated across multiple PEs, i.e., each PE receives a same chunk of three weight elements in the first iteration, and receives different chunk of weight elements in the next iterations. This particular replicated and temporal distribution can be captured with our temporal map directive with size and offset parameter being 3, resulting in `TemporalMap(3,3) S`, where `S` is the first dimension of the weight data structure. Putting it together, spatial map on `X'` followed by a temporal map on `S` captures data mapping and movement behavior across PEs and time corresponding to the two loops in the loop-nest version, and these two directives are enclosed in the red box in Figure 4(c). Each data-centric representation is a complete description of a unique dataflow.

3.2 Dataflow Playground

We build six example dataflows upon the simple 1D convolution discussed in Figure 4 (d) to demonstrate how small changes to a dataflow expose various forms of reuse—both spatial and temporal. Figure 5 illustrates those six example dataflows, which consists of a base dataflow Figure 5(A) and its variants. We modify the directive order, spatially/temporally mapped dimensions, mapping size, and PE clustering and discuss their impact on data reuse.

Directive Order. A change in directive order can result in an entirely different temporal reuse (or, stationary behavior). For example, the sequence of directives in mapping in Figure 5(A) indicates that all data indices of `S` should be explored before working on the next chunk of `X'` indices. This order results in temporally reusing values of data corresponding to `X'` indices (i.e., partial sums) for all indices of `S`. Therefore, this dataflow is informally referred to as

output-stationary and partitioned across multiple outputs in parallel. Figure 5(B) shows the impact of interchanging the order of directives. This results in a weight-stationary dataflow, because PEs can temporally reuse weight values corresponding to S indices, for all indices of X' before going to next chunk of S indices. Similarly, Figure 5(C) and (D) shows the spatial distribution on S instead of X', and also the impact of data movement order on temporal reuse leading to different dataflow variations. This indicates why the informal dataflow name should not be taken as a complete and precise specification of its behavior.

Spatially and Temporally Mapped Dimensions. In Figure 5(A) the directive `SpatialMap(1,1) X'` (where X' refers to the first dimension of the output data structure), spatially distributes indices of the X' dimension with a chunk size of one (the `size` parameter) across PEs with an offset of one (the `offset` parameter). This means that each PE works on a different column of the output data space. If the number of PEs is not sufficient to cover all indices of the dimension mapped, then the mapping is folded over time across the same set of PEs. Also, if `offset` value is smaller than `size` value, then there will be an overlap of indices across consecutive PEs, and this is useful in describing mappings on input activation dimensions X and Y because their iteration space is skewed.

Similarly, `TemporalMap(1,1) S` (where S refers to the first dimension of filter weight data structure), distributes indices of the S dimension with a chunk size of one across time steps with an offset of one. This means that each PE works on the same column of the weight data space. Since all PEs get the same data indices corresponding to a temporally mapped dimension, this creates an opportunity for *spatial reuse*, i.e., multicasting the same data values across PEs in a time step.

Mapping Size. In all of the mappings from Figure 5A-D, the mapping sizes (first argument) of weights and outputs are one – resulting in full temporal reuse of weights but no temporal reuse of outputs (e.g., mapping B and D) or vice versa (e.g., mapping A and C). There is no temporal reuse of inputs in any mapping. Increasing the map size of the spatial or temporal maps can help in presenting opportunities for partial temporal reuse, which can capture convolutional reuse of inputs in CNN layers. For example, the spatial map corresponding to the S dimension in Figure 5(E) helps in exploiting the partial temporal reuse of input data across time steps.

PE Clustering for Multi-dimensional Spatial Distributions. As can be seen in Figure 5(A-E), data mappings related to a map in the outer position get updated after a full exploration of a map in the inner position. This inherent assumption can limit certain dataflow behaviors where one might be interested in simultaneously exploiting spatial distribution of more than one data dimensions.

To address this, we introduce another directive called *Cluster* as a mean to support the simultaneous spatial distribution of multiple data dimensions. The cluster directive logically groups multiple PEs or nested sub-clusters (when a dataflow has multiple cluster directives) of `size` parameter. For example, CLUSTER (3) in Figure 5(F) arranges available PEs into groups of three, resulting in two clusters of three PEs.

All the mapping directives specified above a CLUSTER directive perform the mapping across logical clusters created by the CLUSTER directive. All the mapping directives specified below a CLUSTER directive perform the mapping across PEs or lower level logical clusters inside a logical cluster created by the CLUSTER directive. That is, all the mapping directives above a CLUSTER directive see logical clusters while those below the CLUSTER directive see *inside* of each logical cluster. With this mechanism, one can specify complex dataflows with multiple parallelization dimensions represented by multiple SPATIALMAP directives (one in each cluster level). An example of this can be seen in Figure 5(F), where the X' dimension is spatially distributed across clusters, and the S dimension is spatially distributed within the cluster. The cluster directives enable us to represent existing real-world accelerator dataflows, such as Eyeriss [11] since it involves the spatial distribution of R and Y dimensions simultaneously, and also NVDLA [1] which involves the spatial distribution of K and C dimensions. Another advantage of the cluster directive is that its notion of grouping multiple PEs can represent coarse-grained PEs in accelerators, such as SIMD units [43] and matrix tensor accelerators like GPU Tensor Cores.

In summary, we discussed five transformations that capture all possible aspects of dataflows: scheduling, tiling, and mapping. As shown in Figure 5 the data-centric directives can concisely represent all of those aspects. We envision that the data-centric representation could be either auto-generated from a loop nest version of the dataflow (with affine constraints), or manually written.

3.3 Hardware Implications of Reuse

As we discussed above, various data reuse opportunities appear based on the dataflow. Table 1 summarizes how such opportunities appear in the relationship of spatially mapped dimension within a cluster (Map column) and inner-most temporally mapped dimension (InnerMap column). For example, if output channels (K) are spatially mapped, a decoupled data structure, input feature map, does not change over space. That is, all the PEs receive the same input feature map, which implies a full spatial reuse opportunity (broadcast). In the same example, when the inner-most temporally mapped dimension is the input channels (C), the input channel changes every iteration, which provides temporal reduction opportunities of outputs.

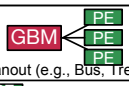
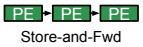
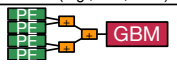
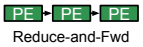
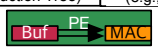
Although a dataflow provides temporal or spatial data reuse opportunities, appropriate hardware support is required to actually exploit these phenomena. Table 2 summarizes four reuse categories and corresponding hardware implementation to support them. As the table shows, reuse can be either spatial or temporal. Based on the data structure, the communication type can be either multicast (input tensors) or reduction (output tensors). Multicast is a communication type that delivers the same data to multiple targets over space (different PEs at the same time) or time (the same PE in different time). Therefore, multicast is one to many communication type, which requires either a fan-out network-on-chip structure such as bus or tree, or a "stationary" buffer to hold the data and deliver it to the future. In contrast, the reduction is many to one communication type, which applies to partial sums to generate final outputs. The reduction also can be either spatial or temporal. Example hardware to support spatial reduction is a reduction tree or reduce-and-forward chain such as systolic arrays. Temporal reduction can be supported by a read-modify-write buffer.

In summary, different dataflows (expressed via our directives) expose different forms of reuse: spatial and temporal, both for multicasts and reductions, which in turn can have multiple hardware

Table 1: Reuse opportunities based on spatially-mapped dimensions in combination with innermost temporally-mapped dimensions. Filters (F), Inputs (I), and Outputs (O) are considered separately. For brevity, X/Y should be interpreted as X'/Y' as appropriate.

Spatial							Temporal						
Mapped Dim.	Coupling			Reuse Opportunity			Innermost Mapped Dim.	Coupling			Reuse Opportunity		
	F	I	O	F	I	O		F	I	O	F	I	O
K	✓		✓		Multicast		C	✓	✓				Reduction
							R/S	✓		✓		Multicast	
							X/Y		✓	✓	Multicast		
C	✓	✓				Reduction	K	✓		✓		Multicast	
							R/S	✓		✓		Multicast	
							X/Y		✓	✓	Multicast		
R/S	✓		✓		Multicast		K	✓		✓		Multicast	
							C	✓	✓				Reduction
							X/Y		✓	✓	Multicast		
X/Y		✓	✓	Multicast			K	✓		✓		Multicast	
							C	✓	✓				Reduction
							R/S	✓		✓		Multicast	

Table 2: Hardware Implementation Choices for supporting spatial and temporal reuse. Note - by *temporal multicast*, we refer to *stationary* buffers from which the same data is read over time.

Reuse Type	Communication Type	HW Implementation Choices	
Spatial	Multicast	 Fanout (e.g., Bus, Tree)	 Store-and-Fwd (e.g., Systolic Array)
	Reduction	 Fanin (e.g., Reduction Tree)	 Reduce-and-Fwd (e.g., Systolic Array)
Temporal	Multicast	 Multiple reads from a buffer	
	Reduction	 Multiple read-modify-write to a buffer	

implementations. Reasoning about dataflows in this structured manner exposes new insights and potential microarchitectural solutions. The discussion so far focused on a simple 1D convolution, which itself exposed many possible dataflows and reuse opportunities. We extend this to a full convolution loop and analyze reuse opportunities within a specific dataflow.

3.4 Extended Example: Row-stationary Dataflow

Figure 6 presents detailed mapping and reuse patterns across two unit time steps of an example row-stationary dataflow [11] over a six-PE accelerator. The accelerator has two PE clusters with three PEs in each cluster. We use the same example layer previously used in Figure 1. Figure 6(a) and (b) are compute- and data-centric representations of the row-stationary dataflow. Figure 6(c) shows how the mapping moves across space (PE clusters) and time Figure 6(d) shows the actual coordinates of each tensor across two time steps and two clusters (i.e., time and space). Each colored box in Figure 6(d) represents replicated data points, which imply reuse opportunities. Based on the replicated data points, we can infer data reuse over the PE array, as shown in data reuse row in Figure 6(d). The mapping in Figure 6(d) shows that the same set of input activation values are replicated across two clusters in a skewed manner within the same time step, which implies spatial reuse opportunities in the diagonal direction of the example PE array. Similarly, Figure 6(d) shows that the same set of weight values are replicated over two time steps within the same PE, which implies temporal reuse opportunities and weight-stationary style dataflow in unit time step granularity. Note that the dataflow is still row-stationary in a coarse-grained time step although it is weight stationary in unit time steps we define in Figure 6 (a) and (b). Finally, Figure 6 (d) shows the same set of output activation over PEs in each PE cluster, which means that all the PEs in each cluster cooperate to generate a set of output activation data. That is, each PE in a PE cluster generates different partial sums for the same output activation, and they need to be accumulated across PEs in each PE cluster to generate final output activation values.

Based on the example analysis in Figure 6, we observe that the data reuse pattern exactly matches the original work [11]: reuse in the horizontal direction for filter weights and vertical for outputs (partial sum accumulation), and reuse in the diagonal direction for input activations.

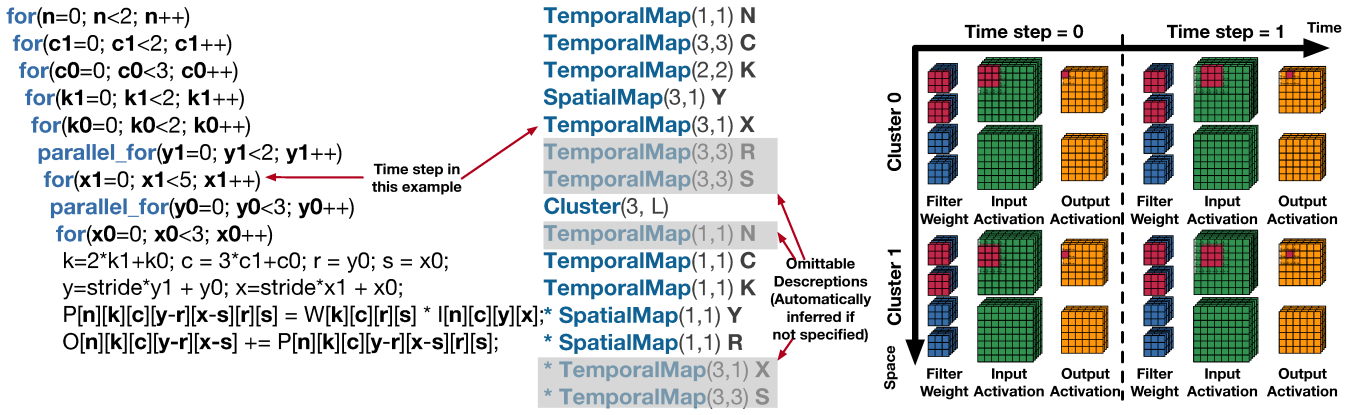
In summary, reuse opportunities are based on the replicated data across time or space (PEs), which implies temporal and spatial reuse opportunities, respectively. The examples in this section demonstrate the need for a fast, accurate quantitative methodology to compute reuse for complex dataflows.

4 QUANTITATIVE DATAFLOW ANALYSIS

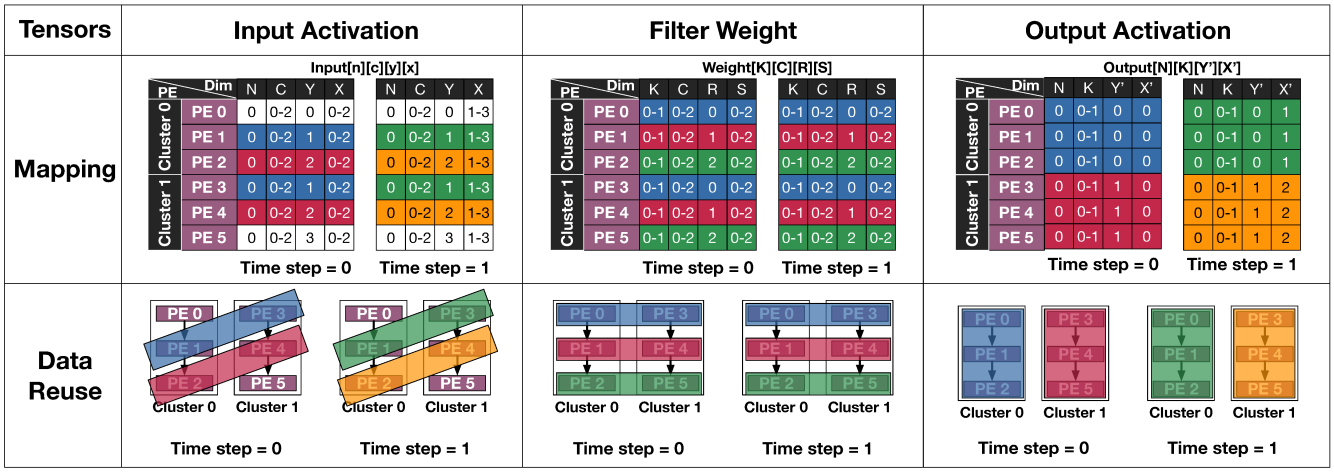
In this section, we present our approach to quantitatively estimating runtime and energy efficiency of dataflows on a target DNN model and hardware configuration. Based on the approach, we implement an analysis framework, MAESTRO, which consists of five engines: tensor, cluster, reuse, performance analysis, and cost analysis. Figure 7 provides a high-level overview of the five engines. In the interest of space, we only discuss high-level algorithms without edge case handling, multiple layers, and multiple cluster levels. For details, we present them in our open-source repository [2].

4.1 Preliminary Engines

Tensor Analysis. As described in Figure 7, the tensor analysis engine identifies dimension coupling for each tensor based on specified layer operations. For example, in depth-wise convolutions, output activation is not coupled with the output-channel dimension but coupled with the input channel dimension. Note that depth-wise convolution can be understood either in this manner or by eliminating input channel dimension (C). We select this convention because it aligns with MAESTRO's input-centric cost model. MAESTRO allows users to specify tensors with arbitrary dimension coupling,

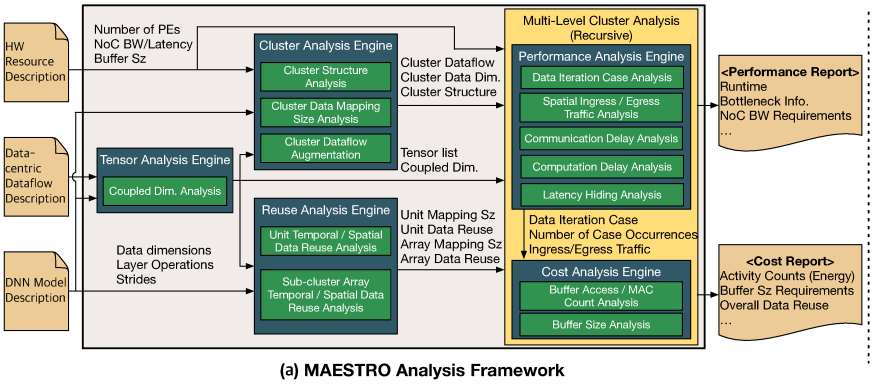


(a) Loop nest Representation (b) Data-centric Representation (c) Visualized data mapping over time



(d) Data mapping and reuse of each tensor

Figure 6: An extended example of a row-stationary style dataflow mapped on a six-PE accelerator. We select our own tile sizes for any not specified in the original work [11]. We do not apply additional mapping optimizations to minimize PE under-utilization. Colors represent data replication either across time or space (PEs). Directives with asterisks indicate fully unrolled directives that cover entire data dimension with one mapping.



(a) MAESTRO Analysis Framework

Engine	Analysis Description
Tensor Analysis	- Extract tensor names, tensor types (input/output), coupled dimensions, and size of each coupled dimenisons
Cluster Analysis	- Extract dataflow directives for each cluster level - Extract data dimension sizes for each cluster level - Infer missing directives, apply stride, and so on - Analyze the number of sub-clusters at each cluster level
Reuse Analysis	- Analyze unit mapping size and the amount of reuse across unit steps for every possible combination of data iteration position cases. (***computes temporal reuse***). - Consolidate unit step reuse analysis results to compute reuse over entire sub-cluster array for each cluster level. (***computes spatial reuse***).
Performance Analysis	- Identifies all the possible data iteration cases and the number of occurrences of the cases - Computes ingress/egress traffic considering data reuse - Computes communication (ingress/egress) and computation delay - Considering latency hiding, determine outstanding delay
Cost Analysis	- For each data iteration case, computes the number of buffer accesses and MAC operations - Identify buffer size requirements based on worst-case analysis

(b) Analysis in Each Engine

Figure 7: An overview of MAESTRO's analysis framework. For simplicity, we omit components other than analysis engines.

```
Inputs: A tensor information table (tensor_tbl), a cluster information table
(cluster_info_tbl), a mapping information table with mapping and reuse size for all
the possible data iteration position cases for each cluster level (mapping_info_tbl),
an abstract hardware model of the target DNN accelerator(hw_model).
Output: Statistics of the target DNN, accelerator, and dataflow (stats)
Object: To compute the mapping size and the amount of data reuse for all the
possible data iteration position cases.
Procedure PerformanceAndCostAnalysisEngine:
 Initialize(stats);
 /* Extracts the cross product of all the possible data iteration cases
    (Init, Steady, and Edge) of each data dimension */
 iteration_cases = ExtractDataIterationCases(tensor_tbl , cluster_info_tbl);
 for each iter_case in iteration_cases
  num_case_occurrences = GetNumCaseOccurrences(iter_case, tensor_tbl,
                                                          cluster_info_tbl);
  /* Considering iteration case, compute the number of partial sums for each PE */
  num_psums = GetNumPSums(iter_case, cluster_info_tbl , mapping_info_tbl);
  /* Considering reuse and iteration case, compute the amount of new input tensor
     data to be fetched from a buffer in upper cluster levels */
  cluster_ingress_traffic = GetNumPSums(iter_case, tensor_tbl ,
                                        cluster_info_tbl , mapping_info_tbl);
  /* Considering reuse and iteration case, compute the amount of output tensor
     data to be committed to a buffer in upper cluster levels */
  cluster_egress_traffic = GetNumOutputs(iter_case, tensor_tbl ,
                                        cluster_info_tbl , mapping_info_tbl);
  //// Core cost analysis ////
  for each tensor in tensor_tbl
   stats.upstream_buffer_read[tensor] += cluster_ingress_traffic[tensor];
   stats.downstream_buffer_write[tensor] += cluster_ingress_traffic[tensor];
   stats.upstream_buffer_write[tensor] += cluster_egress_traffic[tensor];
   stats.downstream_buffer_read[tensor] += num_psums;
   stats.upstream_buffer_size_req[tensor] =
                                  2*Max(stats.upstream_buffer_size_req[tensor],
                                        cluster_ingress_traffic[tensor],
                                        cluster_egress_traffic[tensor]);
   stats.downstream_buffer_size_req[tensor] =
                                 2*Max(stats.downstream_buffer_size_req[tensor],
                                       num_psums, cluster_egress_traffic[tensor]);
  end
  //// Core performance analysis ////
  ingress_delay = GetDelay(cluster_ingress_traffic, hw_model);
  egress_delay = GetDelay(cluster_output_traffic, hw_model);
  compute_delay = GetComputeDelay(num_psums, hw_model);
  compute_delay += GetPSumFwdDelay(iter_case, tensor_tbl,
                                   cluster_info_tbl, mapping_info_tbl);
  /* Considers double-buffering; treats the initialization case as an exeception */
  if IsFullInit(iter_case) then outstanding_delay = ingress_delay
                                                  + compute_delay + egress_delay;
  else
   outstanding_delay = Max(ingress_delay, egress_delay, compute_delay);
  end
  stats.run_time  += outstanding_delay * num_case_occurrences;
  stats.num_macs += num_psums * num_active_clusters * num_case_occurrences;
 end
 Return(stats);
endprocedure
```

Figure 8: A high-level overview of algorithms in performance and cost analysis engines.

and such coupling relationship is input to the rest of engines, which provides generality to MAESTRO.

Cluster Analysis. A PE cluster refers to a group of PEs that processes one or more data dimensions in parallel, specified by the CLUSTER directive. Figure 7 (b) describes the analysis in Cluster Analysis (CLA) engine. The CLA engine analyzes a given dataflow description written in dataflow directives to identify the number of sub-clusters, extract cluster dataflow directives and data dimensions, and augment the given dataflow descriptions for missing directives, stride handling, and so on, for each cluster level.

Reuse Analysis. Figure 7 (b) includes a high-level description of analysis in data reuse analysis (RA) engine. RA engine identifies the amount of temporal and spatial reuse across adjacent time steps, which is the data iteration corresponding to the inner-most non-temporally/spatially unrolled mapping directive.

4.2 Performance Analysis

Figure 7 (a) presents a high-level overview of the performance and cost analysis engine, and Figure 8 shows high-level algorithm of the performance analysis (PA) engine. Utilizing the reuse information computed in the RA engine, PA engine computes the runtime for all the possible cases based on the data dimension and dataflow. The computed runtime is multiplied with the number of each case's occurrences and accumulated to compute the total runtime. The runtime of a DNN accelerator consists of communication delay (L2 to L1, L1 to L2, local forwarding) and computation delay in each PE, which are directly related to the accelerator's hardware parameters. PA engine considers double buffering when it computes the outstanding delay (the worst case delay of communication/computation delay) that directly contributes to the runtime.

To estimate communication delays, MAESTRO relies on its analytical network-on-chip (NoC) model based on a pipe model similar to other analytic models [30]. The pipe model utilizes two parameters, the pipe width (bandwidth) and length (average delay), to estimate the communication delay via NoC. The model incorporates a pipelining effect as many packet-switching NoCs have similar behavior. Various combinations of the bandwidth and average delay enables to model NoC structures with reasonable accuracy. For example, Eyeriss [11] has a two-level hierarchical bus with dedicated channels for input, weight, and output tensors. Therefore, a bandwidth of 3X properly models the top level NoC. The average latency depends on implementation details; users should choose an appropriate value considering implementation details (e.g., the use of ingress/egress buffers, which adds one cycle delay each). For more complicated NoC architectures, users should select bisection bandwidth and average latency considering uniform communication to all the PEs from a global buffer. For example, a $N \times N$ 2D mesh network with the injection point at one of the corners, the bisection bandwidth is N, and the average latency is N. Assuming that the user has access to the NoC implementation information, the NoC model is precise when the NoC is a bus or a crossbar.

4.3 Cost Analysis

Figure 8 describes how the cost analysis (CA) engine computes the number of buffer accesses and estimates the buffer size requirements for each tensor, considering data reuse computed in the RA engine and data iteration cases. Utilizing the access counts and the number of MAC operation information, MAESTRO computes the energy cost. MAESTRO includes an energy model based on those activity counts and Cacti [29] simulation, which can be replaced by any other energy model based on such activity counts (e.g., Accelergy [47]).

4.4 Complex Dataflow Analysis

Multi-cluster Analysis. Multi-cluster cases can be split into single-cluster cases with the data dimension size set as the mapping size of the corresponding mapping directive in the upper cluster. The outstanding delay of a cluster level becomes the computation delay of the next cluster level above. To handle various edge cases that affects all the lower cluster levels, MAESTRO recursively performs performance and cost analysis, as illustrated in Figure 7. In the recursive analysis, the base case is the inner-most cluster whose sub-clusters are actual PEs. Although MAESTRO performs recursion,

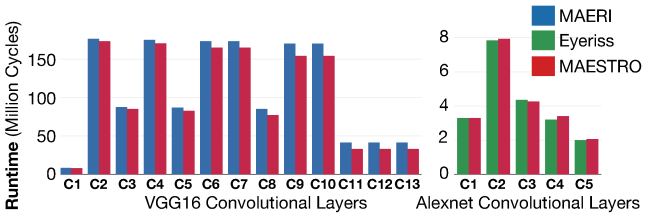


Figure 9: Runtime model validation against MAERI [24] RTL simulation with 64 PEs and Eyeriss [13] runtime reported in the paper with 168 PEs.

the complexity is not high because the number of PE cluster levels are typically two or three. Note that each of the edge cases at each cluster level also needs to be recursively processed. However, in most cases, we observe the number of edge cases across cluster levels is less than 20, which is still in a tractable scale.

Other DNNs. Although we used dense convolution as examples for simplicity, MAESTRO can model a variety of layers (LSTM hidden layer, pooling, fully-connected, transposed convolution, and so on) based on the generality of the data-centric approach. Our data-centric approach supports all the operations represented as the loop nest with two input tensors and one output tensor wherein all the tensor indices are coupled in only one or two data dimensions in affine functions. MAESTRO also can model uniformly distributed sparsity for any supported dataflow. Support for more complex statistical sparsity distributions is future work.

4.5 Model Validation

We validated MAESTRO's performance model against RTL simulations of two accelerators - MAERI [24] and Eyeriss [13] when running VGG16 and AlexNet respectively[1]. Figure 9 shows that the runtime estimated by MAESTRO are within 3.9% absolute error of the cycle-accurate RTL simulation and reported processing delay [13] in average.

5 CASE STUDIES

Table 4 summarizes the features of frequently used DNN operators from state-of-the-art DNN models [6, 18, 28, 34, 35]. Early and late layers refer to layers with high-resolution activation with shallow channels and vice versa, respectively. We label them as early and late layers because such layers appear early and late in classification networks [18, 28, 35, 42]. We compare the number of input channels and the input activation height to identify them[2].

With MAESTRO, we perform deeper case studies about the costs and benefits of various dataflows when they are applied to different DNN operations listed in Table 4. We evaluate five distinct dataflow styles listed in Table 3 in Section 5.1 and the preference of each dataflow to different DNN operators. For energy estimation, we multiply activity counts with base energy values from Cacti [29] simulation (28nm, 2KB L1 scratchpad, and 1MB shared L2 buffer). We also present distinct design space of an early layer (wide and shallow) and a late layer (narrow and deep) to show the dramatically different hardware preference of different DNN operator styles and dataflow in Section 5.2.

[1]MAERI RTL is open-source. For Eyeriss, we use the reported runtime for AlexNet because detailed mapping parameters are described for only AlexNet in the paper.
[2]If C > Y, late layer. Else, early layer

Table 3: Five example dataflows used for the evaluation. For conciseness, we omit redundant directives that are automatically inferred by MAESTRO. YX-P, YR-P, and CK-P dataflows are motivated by Shidiannao [15], Eyeriss [11], and NVDLA [1], respectively. The name of each dataflow is based on spatial dimensions from the upper-most cluster level.

Partitioning Strategy	Dataflow	Characteristics
C-Partitioned (C-P)	TemporalMap (1,1) K TemporalMap (Sz(R),1) Y TemporalMap (Sz(S),1) X TemporalMap (Sz(R),Sz(R)) R TemporalMap (Sz(S),Sz(S)) S SpatialMap (1,1) C	- Large spatial reduction opportunities (Large output activation reuse) - Small input activation/filter reuse - Input channel (C) parallelism - No local reuse
X-Partitioned (X-P)	TemporalMap (1,1) K TemporalMap (1,1) C TemporalMap (Sz(R),Sz(R)) R TemporalMap (Sz(S),Sz(S)) S TemporalMap (Sz(R),1) Y SpatialMap (Sz(S),1) X	- Large temporal reuse of filter - Spatial reuse opportunities (via halo in input activation) - Input column (X) parallelism - Weight-stationary
YX-Partitioned (YX-P)	TemporalMap (1,1) K SpatialMap (Sz(R),1) Y TemporalMap (8+Sz(S)-1,8) X TemporalMap (1,1) C TemporalMap (Sz(R),Sz(R)) R TemporalMap (Sz(S),Sz(S)) S Cluster(8) SpatialMap (Sz(S),1) X	- Large temporal reuse of filter - Better spatial reuse opportunities over X-P (via 2D halo in input activation) - 2D activation (X and Y) parallelism - Output-stationary - Motivated by Shi-diannao [14]
YR-Partitioned (YR-P)	TemporalMap (2,2) C TemporalMap (2,2) K SpatialMap (Sz(R),1) Y TemporalMap (Sz(S),1) X TemporalMap (Sz(R),Sz(R)) R TemporalMap (Sz(S),Sz(S)) S Cluster(Sz(R)) SpatialMap (1,1) Y SpatialMap (1,1) R	- Large temporal reuse of input activation and filter - Spatial reduction opportunities (spatial reuse of output activations) - Activation row (Y) and filter column (S) parallelism - Row-stationary - Motivated by Eyeriss [10]
KC-Partitioned (KC-P)	SpatialMap (1,1) K TemporalMap (64,64) C TemporalMap (Sz(R),Sz(R)) R TemporalMap (Sz(S),Sz(S)) S TemporalMap (Sz(R),1) Y TemporalMap (Sz(S),1) X Cluster(64) SpatialMap (1,1) C	- Spatial reuse of input activation - Large spatial reduction factor (64-way) over input channel (C) - Input/output channel (C and K) parallelism - Weight-stationary - Motivated by NVDLA [1]

5.1 Case study I: Dataflow Trade-offs

Figure 10 shows the DNN-operator granularity estimation of runtime and energy of each dataflow across five state-of-the-art DNN models listed in Section 5. Note that this should be considered a comparison of dataflows—not of actual designs, which can contain several low-level implementation differences, e.g., custom implementations of logic/memory blocks, process technology, and so on. We observe that KC-P dataflow style dataflow provides overall low runtime and energy. However, the energy efficiency in VGG16 (Figure 10 (b)) is worse than YR-P (Eyeriss [11] style) dataflow, and the runtime is worse than YX-P (Shidiannao [15] style) dataflow in UNet (Figure 10 (e)). This is based on the different preference toward dataflow of each DNN operator. YX-P provides short runtime to segmentation networks like UNet, which has wide activation (e.g., 572x572 in the input layer) and recovers the original activation dimension at the end via up-scale convolution (e.g., transposed convolutions). Such a preference to the YX-P style is mainly based on its parallelization strategy: it exploits parallelism over both of row and column dimensions in activation. The energy efficiency of YR-P dataflow in VGG16 is based on its high reuse factor (the number of local accesses per fetch) in early layers, as shown in red bars in Figure 11 (a) and (b) (note the log scale). The YR-P dataflow has 5.8× and 15.17× higher activation and filter reuse factors, respectively, in early layers. However, in late layers, the reuse factors of YR-P and KC-P dataflow are almost similar (difference < 11%), so the KC-P

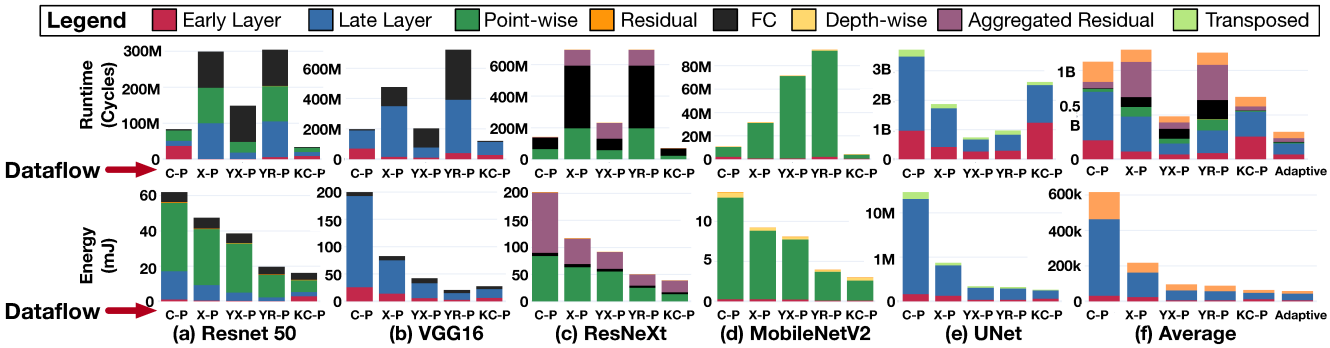


Figure 10: Plots in top and bottom rows present runtime and energy estimation of five dataflows listed in the table, respectively. We apply 256 PEs and 32GBps NoC bandwidth. We evaluate all the dataflows using five different DNN model; Resnet50 [18], VGG16 [42], ResNeXt50 [35], MobileNetV2 [28], and UNet [34]. The final column (f) presents the average results across models for each DNN operator type listed in Table 4 and the adaptive dataflow case.

Table 4: Operators in state-of-the-art DNNs and their features and implication. Bottleneck [18] and depth-wise separable convolution [6] are listetd in a fine-grained operators (point-wise convolution, depth-wise convolution, and residual links). Examples are based on notable networks (VGGnet [42] and DCGAN [4]) and state-of-the-art networks (MobileNetV2 [28], ResNet50 [18], ResNeXt50 [35].

DNN Operators	Examples	Characteristics
CONV 2D Early Layers	- VGG16 CONV1-3 - MobileNetV2 CONV1 - UNet Conv1-2	- Large activation height and width - Shallow input/output channels
CONV 2D Late Layers	- VGG16 CONV4-13 - MobileNetV2 CONV2-3 - UNet Conv3-5	- Small activation height and width - Deep input/output channels
Fully-Connected	- VGG16 FC1-3 - ResNet50 FC1000d - ResNeXt50 FC1000d	- GEMM operation
Point-wise Convolution (1x1 CONV2D)	- ResNet50 CONV2-5 (Bottleneck) - MobileNetV2 Bottleneck 1-7	- No parallelism in filter rows and columns - No convolutional reuse opportunities
Depth-wise Convolution	- MobileNetV2 Bottleneck 1-7	- Reduced computation compared to CONV2D - Reduced data reuse opportunities - Higher NoC BW requirements
Residual Links (Skip Connections)	- ResNet50 CONV2-5 (Bottleneck) - MobileNetV2 Bottleneck1-7	- Extra global buffer / DRAM accesses to fetch previous activation or buffer space for entire activation
Aggregated Residual Blocks	- ResNeXt50 CONV2-5	- More data parallelism via branching structure - Concatenation and reduction of activation required
Transposed Convolution	- UNet UpConv 1-4 - DCGAN CONV1-4	- Upscaled output activations - Structured sparsity in output activations

dataflow provides similar energy efficiency as YR-P in these cases. This can also be observed in the late layer (blue) bars in Figure 10 bottom-row plots.

Although the KC-P and YX-P dataflows provide low runtime (Figure 10), it comes with high NoC cost, as the high bandwidth requirements shown in Figure 11 (c) highlight. Based on the operator type, some dataflows require dramatically higher NoC bandwidth than others. For example, YX-P requires high bandwidth for point-wise convolution as it has no convolutional reuse (i.e., overlapped activation data points among sliding windows) because of its 1x1

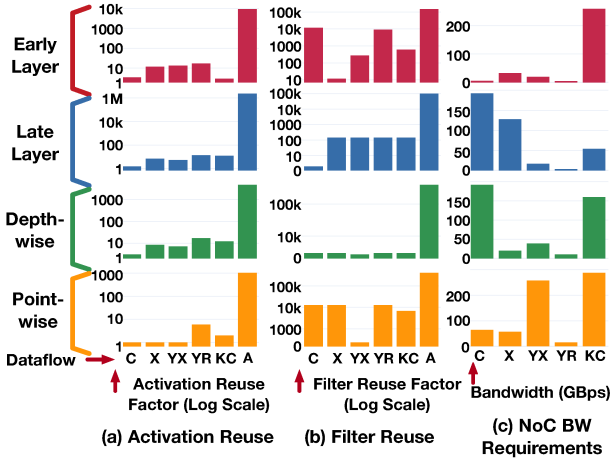


Figure 11: Reuse and NoC bandwidth requirements of dataflows in Table 3 with 256 PEs for four common DNN operators from Table 4. We select representative operators from state-of-the-art DNN models (Early layer: CONV1 in Resnet50 [18], late layer: CONV13 in VGG16 [42], depth-wise convolution (DWCONV): DWCONV of CONV2 in ResNeXt50 [35], point-wise convolution: first conv of bottleneck1 in MobilenetV2 [28] C, X, YX, YR, and KC refers to C-P, X-P, YX-P, YR-P, and KC-P dataflows. A refers to algorithmic maximum reuse.).

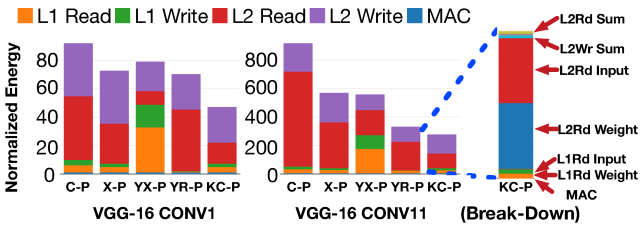


Figure 12: The breakdown of energy consumption (MAC and L1/L2 scratchpad access energy) of the dataflows from Table 3. The access counts generated by MAESTRO are multiplied by appropriate energy values from Cacti [29]. The values are normalized to the MAC energy of C-P.

kernel while YX-P is optimized to exploit convolutional reuse via spatial reuse.

The diverse preference to dataflows of different DNN operators motivates us to employ optimal dataflow for each DNN operator type. We refer such an approach as adaptive dataflow and present the benefits in Figure 10 (f), the average case analysis across entire models in DNN operator granularity. By employing the adaptive approach, we could observe a potential 37% runtime and 10% energy reduction. Such an optimization opportunity can be exploited by flexible accelerators like Flexflow [25] and MAERI [24] or via heterogeneous accelerators that employ multiple sub-accelerators with various dataflow styles in a single DNN accelerator chip.

5.2 Case study II: Hardware Design-Parameters and Implementation Analysis

Using MAESTRO, we implement a hardware design space exploration (DSE) tool that searches four hardware parameters (the number of PEs, L1 buffer size, L2 buffer size, and NoC bandwidth) optimized for either energy efficiency, throughput, or energy-delay-product (EDP) within given hardware area and power constraints. The DSE tool receives the same set of inputs as MAESTRO with hardware area/power constraints and the area/power of building blocks synthesized with the target technology. For the cost of building blocks, we implement float/fixed point multiplier and adder, bus, bus arbiter, and global/local scratchpad in RTL and synthesis them using 28nm technology. For bus and arbiter cost, we fit the costs into a linear and quadratic model using regression because bus cost increases linearly and arbiter cost increases quadratically (e.g., matrix arbiter).

The DSE tool sweeps a target design space specified in the range of each parameter and search granularity. However, it skips design spaces at each iteration of hardware parameters by checking the minimum area and power of all the possible design points from inner loops of hardware parameters. This optimization allows it to skip invalid design points in a various granularity that reduces a large number of futile searches, which led to a large effective DSE rate ranging from 3.3K to 0.46M designs per second, as presented in Figure 13 (c). Figure 13 (c) shows statistics of four DSE runs explored the design space. We ran DSEs on a machine with i7-8700k CPU and 32GB memory operating Linux Mint 19 OS. We run four sets of the DSE on the machine at the same time, and all of them terminated within 24 minutes, with effective DSE rate of 0.17M designs per second, on average.

Design Space Analysis: Using the DSE tool, we explore the design space of KC-P and YR-P dataflow accelerators. We set the area and power constraint as 16mm^2 and 450mW, which is the reported chip area and power of Eyeriss [13]. We plot the entire design space we explored in Figure 13.

Whether an accelerator can achieve peak throughput depends on not only the number of PEs but also NoC bandwidth. In particular, although an accelerator has sufficient number of PEs to exploit the maximum degree of parallelism a dataflow allows, if the NoC does not provide sufficient bandwidth, the accelerator suffers a communication bottleneck in the NoC. Such design points can be observed in the area-throughput plot in Figure 13 (a). YR-P dataflow requires low NoC bandwidth as shown in Figure 11 (c) so it does not show the same behavior as KC-P dataflow. However, with more stringent area and power constraints, YR-P dataflow will show the same behavior.

During DSE runs, MAESTRO reports buffer requirements for each dataflow and the DSE tool places the exact amount buffers MAESTRO reported. Contrary to intuition, larger buffer sizes do not always provide high throughput, as shown in buffer-throughput plots in Figure 13 (plots in the second column). The optimal points regarding the throughput per buffer size are in the top-left region of the buffer-throughput plots. The existence of such points indicates that the tiling strategy of the dataflow (mapping sizes in our directive representation) significantly affects the efficiency of buffer use.

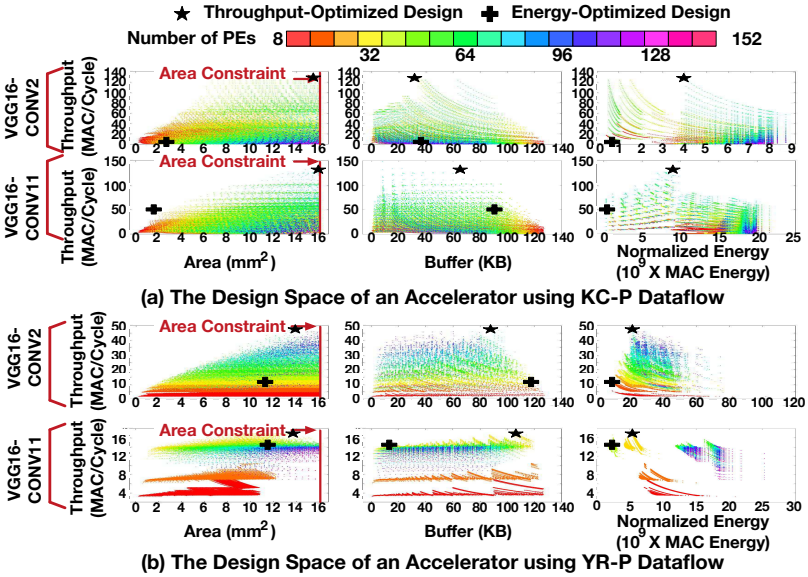
We also observe the impact of hardware support for each data reuse, discussed in Table 2. Table 5 shows such design points found in the design space of KC-P dataflow on VGG16-conv2 layer presented in the first row of Figure 13 (a). The first design point is the throughput-optimized design represented as a star in the first row of Figure 13. When bandwidth gets smaller, the throughput significantly drops, but energy remains similar. However, the lack of spatial multicast or reduction support resulted in approximately 47% energy increase, as the third and fourth design points shows.

We observe that the throughput-optimized designs have a moderate number of PEs and buffer sizes, implying that hardware resources need to be distributed not only to PEs but also to NoC and buffers for high PE utilization. Likewise, we observe that the buffer amount does not directly increase throughput and energy efficiency. These results imply that all the components are intertwined, and they need to be well-balanced to obtain a highly-efficient accelerator.

6 RELATED WORKS

Hardware DSE and dataflow optimization: Dataflow optimization is one of the key optimization targets in many recent DNN accelerators such as Eyeriss [11], Flexflow [25], SCNN [31], and NVDLA [1]. C-brain [43], Flexflow [25], and analyzed the cost-benefit tradeoff of three dataflows and explored the opportunity of adaptive dataflow based on the tradeoff. Ma et al. [26] also constructed an analytic model for convolutions on FPGAs focusing on three loop transformations; interchange, unroll, and tiling. Although their analytic model provides an intuition for trade-offs of dataflows, the model focus on one dataflow style they propose, does not consider regional spatial reuse, spatio-temporal reuse opportunities in DNN accelerators, and also doesn't consider communication delay in the NoC, which can dominate for dataflows with large tile sizes. Also, the target dataflow is optimized for HLS flow, and it requires expressing using complex annotated loop nest with HLS synthesis directives. Caffeine [48] proposed a full automatic FPGA flow that includes pragma-based annotation in programs, dataflow optimization framework, and DSE for FPGAs based on the analytic model defined over loop tiling and unrolling. However, the dataflow search space is limited due to fixed loop orders; three presets termed straightforward, input-major, and weight-mapping.

Past works related to data-centric approaches: There have been some works related to exploring data-centric approaches [21–23], where the approaches reason about flow of data through memory hierarchy, instead of control-structure centric analysis, for locality-enhancement transformations such as multi-level data blocking [22] and data shackling [21]. But, the above data-centric approaches have



DSE Setting / Stats	KC-P Dataflow		YR-P Dataflow	
	VGG16 CONV2	VGG16 CONV11	VGG16 CONV2	VGG16 CONV11
Number of Valid Design Points	175,861	260,028	319,268	115,209
Number of Explored Design Points	3,936,256	251,920,384	3,555,328	227,540,992
DSE Execution Time (min)	12.20	18.29	23.68	8.22
DSE Rate (designs/sec)	5377.40	229561.13	3239.77	461356.43

(c) DSE Statistics

Figure 13: The design space of an accelerator with (a) KC-P and (b) YR-P dataflow. We highlight the design space of an early and a late layer to show their significantly different hardware preference. We apply the area and power of Eyeriss [13] as area/power constraints to the DSE.(16mm^2, 450mW [13]). The color of each data point indicates the number of PEs. Design points with fewer PEs can be paired with larger buffer sizes, up to the area budget. We mark the throughput- and energy-optimized designs using a star and cross.

Table 5: The impact of multicasting capability, bandwidth, and buffer size. Design points are from the design space of Figure 13 (a) VGG16-CONV2.

Design Point	Num PEs	NoC BW (Data pt/Cycle)	Spatial Reuse Support		Temporal Reuse Support	Throughput (MAC/cycle)	Energy (X MACs)
			Multicast	Reduction	Buffer Size (KB)		
Reference	56	40	Yes	Yes	6.13	48.6	5.26 x 10^9
Small bandwidth	56	24	Yes	Yes	6.13	34.54	5.26 x 10^9
No multcast	56	40	No	Yes	2.26	33.39	7.56 x 10^9
No Sp. reduction	56	40	Yes	No	4.68	32.05	7.77 x 10^9

been explored in the context of driving optimizations for multi-level caches, but not estimating energy or throughput of input kernels precisely. We discuss related work on loop-nest notation and reuse analysis in compilers in Section 2.5.

7 DISCUSSION AND FUTURE WORK

This work is motivated by the observation that co-optimizing DNN accelerator microarchitecture and its internal dataflow(s) is crucial for accelerator designers to achieve both higher performance and energy efficiency. In this work, we introduced data-centric directives to specify DNN dataflows in a compact form and understand data reuse opportunities. We also presented an analytical model called MAESTRO to estimate execution time, energy efficiency, and the hardware cost of dataflows. We evaluated our analytical model relative to the MAERI and Eyeriss accelerators and found our model to be highly consistent with cycle-accurate simulations and reported runtime, which shows the soundness of the analytic model. We provided cases studies about the costs and benefits of dataflow choices over in five state-of-the-art DNN models with a focus on common DNN operators in them, showing diverse preference to dataflow and hardware, which motivates adaptive dataflow accelerator and heterogeneous accelerators. Finally, we also demonstrated the use of MAESTRO for design-space exploration of two dataflows in early and late layers, showing dramatically different hardware preference of each layer. Our DSE tool based on MAESTRO enabled fast DSE based on optimization to skip invalid designs, which led to a high average DSE rate of 0.17M designs per second.

In the future, we plan to leverage MAESTRO to implement a dataflow auto-tuner to find an optimal dataflow on the specified DNN model and hardware configuration. With the optimal dataflow, we plan to extend our infrastructure to automatically generate RTL, facilitating end-to-end DNN acceleration flow.

ACKNOWLEDGEMENT

We thank Joel Emer for insightful advice and constructive comments to improve this work; Vivienne Sze and Yu-Hsin Chen for their insights and taxonomy that motivated this work. This work was supported by NSF Awards 1755876 and 1909900.

REFERENCES

[1] 2017. NVDLA Deep Learning Accelerator. http://nvdla.org.

[2] 2019. MAESTRO project page. https://synergy.ece.gatech.edu/tools/maestro/.

[3] V Aklaghi, Amir Yazdanbakhsh, Kambiz Samadi, H Esmaeilzadeh, and RK Gupta. 2018. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *International Symposium on Computer Architecture (ISCA)*.

[4] Luke Metz Alec Radford and Soumith Chintala. 2015. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *arXiv preprint arXiv:1511.06434* (2015).

[5] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, et al. 2015. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv preprint arXiv:1512.02595* (2015).

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

[7] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noel Pouchet, and P. Sadayappan. 2017. Analytical Modeling of Cache Behavior for Affine Programs. *Proc. ACM Program. Lang.* 2, POPL, Article 32 (Dec. 2017), 26 pages. https://doi.org/10.1145/3158120

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[9] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *International conference on Architectural support for programming languages and operating systems (ASPLOS)*.

[10] 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. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[11] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *International Symposium on Computer Architecture (ISCA)*.

[12] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2018. Eyeriss v2: A Flexible and High-Performance Accelerator for Emerging Deep Neural Networks. arXiv:cs.DC/1807.07928

[13] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138.

[14] Jason Cong and Bingjun Xiao. 2014. Minimizing computation in convolutional neural networks. In *International conference on artificial neural networks (ICANN)*. Springer, 281–290.

[15] 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 *International Symposium on Computer Architecture (ISCA)*.

[16] Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. 2013. Learning hierarchical features for scene labeling. *PAMI* 35, 8 (2013), 1915–1929.

[17] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. Tetris: Scalable and efficient neural network acceleration with 3d memory. *ACM SIGOPS Operating Systems Review* 51, 2 (2017), 751–764.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

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

[20] Andrej Karpathy and Li Fei-Fei. 2015. Deep visual-semantic alignments for generating image descriptions. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[21] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. 1997. Data-centric Multi-level Blocking. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97)*. ACM, New York, NY, USA, 346–357. https://doi.org/10.1145/258915.258946

[22] Induprakas Kodukula and Keshav Pingali. 2001. Data-Centric Transformations for Locality Enhancement. *International Journal of Parallel Programming* 29, 3 (01 Jun 2001), 319–364. https://doi.org/10.1023/A:1011172104768

[23] Induprakas Kodukula, Keshav Pingali, Robert Cox, and Dror Maydan. 1999. An Experimental Evaluation of Tiling and Shackling for Memory Hierarchy Management. In *Proceedings of the 13th International Conference on Supercomputing (ICS '99)*. ACM, New York, NY, USA, 482–491. https://doi.org/10.1145/305138.305243

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

[25] 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 *International Symposium on High Performance Computer Architecture (HPCA)*.

[26] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 45–54.

[27] Mostafa Mahmoud, Kevin Siu, and Andreas Moshovos. 2018. Diffy: a Déja vu-Free Differential Deep Neural Network Accelerator. In *International Symposium on Microarchitecture (MICRO)*.

[28] Menglong Zhu Andrey Zhmoginov Mark Sandler, Andrew Howard and Liang-Chieh Chen. 2019. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *arXiv preprint arXiv:1801.04381* (2019).

[29] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* (2009), 22–31.

[30] 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 *Proceedings of the 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[31] A. Parashar *et al.* 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *International Symposium on Computer Architecture (ISCA)*. 27–40.

[32] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. 2010. Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*. 1–11. https://doi.org/10.1109/SC.2010.14

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

[34] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*. Springer, 234–241.

[35] Piotr DollÃąr Zhuowen Tu Saining Xie, Ross Girshick and Kaiming He. 2017. Aggregated Residual Transformations for Deep Neural Networks. *arXiv preprint arXiv:1611.05431* (2017).

[36] Vivek Sarkar. 1997. Automatic selection of high-order transformations in the IBM XL FORTRAN compilers. *IBM Journal of Research and Development* 41, 3 (1997), 233–264. https://doi.org/10.1147/rd.413.0233

[37] Vivek Sarkar and Nimrod Megiddo. 2000. An analytical model for loop tiling and its solution. In *ISPASS*. 146–153. https://doi.org/10.1109/ISPASS.2000.842294

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

[39] Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. 2014. Oil and Water Can Mix: An Integration of Polyhedral and AST-based Transformations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 287–298. https://doi.org/10.1109/SC.2014.29

[40] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. 2012. Analytical Bounds for Optimal Tile Size Selection. In *Compiler Construction - 21st International Conference, CC 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 101–121. https://doi.org/10.1007/978-3-642-28652-0_6

[41] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[42] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks For Large-Scale Image Recognition. In *International Conference on Learning Representations (ICLR)*.

[43] Lili Song, Ying Wang, Yinhe Han, Xin Zhao, Bosheng Liu, and Xiaowei Li. 2016. C-brain: a deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization. In *Design Automation Conference (DAC)*. 1–6.

[44] Alexander Toshev and Christian Szegedy. 2014. Deeppose: Human pose estimation via deep neural networks. In *Conference on Computer Vision and Pattern*

Recognition (CVPR).

[45] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*. ACM, New York, NY, USA, 30–44. https://doi.org/10.1145/113445.113449

[46] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv preprint arXiv:1609.08144* (2016).

[47] Wu, Yannan N. and Emer, Joel S. and Sze, Vivienne. 2019. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*.

[48] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2018. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2018).