LoopTree: Exploring the Fused-Layer Dataflow Accelerator Design Space

Michael Gilbert, Yannan Nellie Wu, Joel S. Emer, *Fellow, IEEE*, and Vivienne Sze, *Senior Member, IEEE*

***Abstract*—Latency and energy consumption are key metrics in the performance of deep neural network (DNN) accelerators. A significant factor contributing to latency and energy is data transfers. One method to reduce transfers or data is reusing data when multiple operations use the same data. *Fused-layer accelerators* reuse data across operations in different layers by retaining intermediate data in on-chip buffers, which has been shown to reduce energy consumption and latency. Moreover, the intermediate data is often tiled (*i.e.*, broken into chunks) to reduce the on-chip buffer capacity required to reuse the data. Because on-chip buffer capacity is frequently more limited than computation units, fused-layer dataflow accelerators may also recompute certain parts of the intermediate data instead of retaining them in a buffer. Achieving efficient trade-offs between on-chip buffer capacity, off-chip transfers, and recomputation requires systematic exploration of the fused-layer dataflow design space. However, prior work only explored a subset of the design space, and more efficient designs are left unexplored. In this work, we propose (1) a more extensive design space that has more choices in terms of tiling, data retention, recomputation and, importantly, allows us to explore them *in combination*, (2) a taxonomy to systematically specify designs, and (3) a model, LoopTree, to evaluate the latency, energy consumption, buffer capacity requirements, and off-chip transfers of designs in this design space. We validate our model against a representative set of prior architectures, achieving a worst-case 4% error. Finally, we present case studies that show how exploring this larger space results in more efficient designs (*e.g.*, up to a 10× buffer capacity reduction to achieve the same off-chip transfers).**

***Index Terms*—Accelerator, analytical modeling, fusion.**

I. Introduction

Deep neural networks (DNNs) are a dominant approach in various applications, such as computer vision [1], [2], [3], [4], natural language processing [5], [6], [7], [8], speech recognition [9], self-driving cars [10], and others. The ubiquity of DNNs, combined with the large amount of computation and data required in DNN processing, motivates the need for low-latency and energy-efficient DNN processing.

Data transfers are a significant component of energy consumption and latency in DNN processing. An effective method to reduce data transfers is by reusing data. For example, if multiple operations use the same data, data transfers from off-chip buffers can be avoided by retaining the data in an on-chip buffer to be reused for the processing of the operations. Thus, we trade off-chip transfers with the chip area allocated for on-chip buffers.

We can categorize data reuse based on the DNN structure. Structurally, DNNs are made of layers that take input feature maps (fmaps) and filters to produce output fmaps. When the operations that reuse data belong to the same layer, the reuse is an *intra-layer reuse*. When the operations that reuse data belong to the different layers—specifically, the output fmap of the previous layer becomes the input fmap of the next layer (i.e, these fmaps are *intermediate fmaps*)—the reuse is an *inter-layer reuse*.

Many accelerators process DNNs in a *layer-by-layer* fashion [11], [12], [13], [14], [15], where all operations for a layer are performed before the operations for the next layer (see Fig. 1(a) and 1(b)). Because operations from a layer are scheduled close together, layer-by-layer processing is efficient at exploiting intra-layer reuse opportunities. On the other hand, reusing the intermediate fmap between layers requires retaining the entire intermediate fmap in a buffer because the entire intermediate fmap is produced before it is used in the next layer (see Fig. 1(b)). DNN fmaps are often large, and retaining them requires large buffers that often do not fit on-chip. More typically, layer-by-layer accelerators do not exploit inter-layer fmap reuse on-chip. Rather, intermediate fmaps are streamed to and from off-chip buffers between the processing of different layers. But this may be undesirable for two reasons. First, off-chip bandwidth is more limited than on-chip bandwidth and large volumes of off-chip transfers may cause bandwidth bottlenecks, which impacts latency. Second, off-chip transfers cost more energy than on-chip transfers.

The amount of intermediate fmap that needs to be retained to exploit inter-layer reuse can be reduced by *tiling* the layers and scheduling the processing of the tiles such that an intermediate fmap tile (*i.e.*, a chunk of the intermediate

Received 17 May 2024; revised 16 July 2024; accepted 29 August 2024. Date of publication 17 September 2024; date of current version 8 November 2024. The review of this article was arranged by Associate Editor Mingoo Seok. *(Corresponding author: Michael Gilbert.)*

Michael Gilbert and Vivienne Sze are with the Department of Electrical Engineering and Computer Science, School of Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139 USA (e-mail: gilbertm@mit.edu; sze@mit.edu).

Yannan Nellie Wu is with Google, Mountain View, CA 94043 USA (e-mail: nellieywu@gmail.com).

Joel S. Emer is with the Department of Electrical Engineering and Computer Science, School of Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139 USA, and also with NVIDIA, Santa Clara, CA 95051 USA (e-mail: emer@csail.mit.edu).

Digital Object Identifier 10.1109/TCASAI.2024.3461716

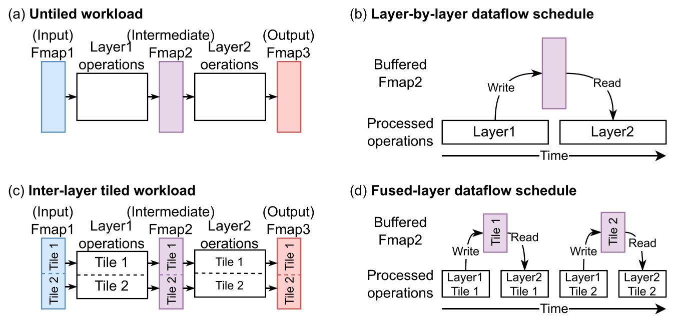

Fig. 1. Comparison of layer-by-layer and fused-layer dataflows. (a) Two layers (white boxes represent operations within the layers) and three fmaps[2]. (b) Layer-by-layer processing produces all of Fmap2 before it is used. (c) Tiling layer operations and fmaps. (d) A fused-layer processing of Layer1 and Layer2, where only a tile of Fmap2 needs to be retained in a buffer at a time.

fmap) can be computed and immediately used between layers (see Fig. 1(c) and 1(d)). When the intermediate fmap tile is not needed anymore, it is released from the buffer. Thus, only a tile of the intermediate fmap needs to be retained at a time. Since this tiling is applied across layers, we refer to the tiling as *inter-layer tiling*. Moreover, we refer to the layers as being *fused* and the accelerators employing them as *fused-layer dataflow accelerators* [16], [17], [18], [19][1].

In DNN accelerators, on-chip buffers often occupy a large share of the chip's area [11], [12], [20]. In contrast, compute units are abundant and use less energy compared to reading from buffers [11], [12], [20]. Fused-layer dataflows can take advantage of this fact by recomputing instead of retaining intermediate data [16], [21], [22].

The choices of tiling, recomputation, and retention are important design aspects that need to be explored in combination for an efficient fused-layer dataflow. However, only a subset of this design space has been explored in prior fused-layer dataflows and design space exploration (DSE) frameworks, leaving more efficient designs unexplored.

In order to precisely discuss the fused-layer dataflow design space, we briefly introduce some nomenclature (more details in Section II-B). Most DNN layers can be viewed as *tensor algebra operations* [23], [24], [25], [26]. Viewed this way, the multidimensional fmaps and weights are represented by tensors, which have multiple ranks corresponding to the dimensions, *e.g.*, channels and width.

Now, we highlight four characteristics of the fused-layer design space, and show the space of choices in prior work in Table I.

- *Partitioned ranks.* Tiling is done by partitioning ranks (*e.g.*, channels and width) in the layer. The more ranks that can be partitioned, the more choices we have to create tiles. However, most prior work only supports a limited set of ranks to partition.
- *Recomputation.* In a given intermediate fmap, there are many ways to choose which activations (*i.e.*, values in an fmap) are recomputed. Most prior works do not support or support only a limited set of recomputation choices.
- *Per-intermediate-fmap recomputation.* A recomputation choice can be made for each intermediate fmap. However, prior work that supports extensive recomputation choices is limited to applying the same recomputation choice for all intermediate fmaps.
- *Per-tensor retention.* Some prior works are limited in choosing the shape of tensor tiles that are retained. Specifically, if a rank is partitioned in the retained tile of a tensor (*i.e.*, a filter or fmap), retained tiles of other tensors also have to partition that rank in the same way. However, being able to make this partitioning choice per tensor has been shown to significantly increase efficiency [21], [27].

Finally, this paper presents the following contributions:

(1) We identify a design space that supports a more extensive set of tiling, recomputation, and retention choices, and their combinations. Our results show that exploring these choices in combination leads to more efficient designs.

(2) We propose a taxonomy to systematically specify designs in our design space in Section III. To describe tiling in our taxonomy, we build on concepts from the Einsum notation [23], [32] (reviewed in Section II-B). We also discuss a fundamental relationship between data retention, reuse, refetch, and recomputation that allows us to simplify the design space while expanding the space of recomputation choices.

(3) We present a model, LoopTree, that supports our design space and validate it. LoopTree (discussed in Section IV), evaluates the latency, energy, amount of off-chip transfers, and required buffer capacity of a given design from our design space. In Section V, we validate this model against a representative set of prior architectures and show a worst-case 4% error.

(4) Using LoopTree, we present case studies that illustrate insights into the design of efficient fused-layer dataflows in Section VI. We show that LoopTree can be used to explore the trade-off between off-chip transfers, buffer capacity, and recomputation. We will also discuss the impact of each design choice, how they interact, and how the shape of DNN layers impacts which design is more efficient.

(5) The case studies provide insights on the impact of design choices and how to perform systematic exploration of the design space. *E.g.*, based on the results and their interpretation in Section VI-B, we highlight how to choose tiling and scheduling choices in order to reduce the required on-chip buffer capacity to achieve the most data reuse.

II. Background and Motivation

In this section, we briefly review relevant concepts in deep neural network (DNN) layers, the Einsum notation, layer fusion, and limitations in prior dataflows and DSE frameworks.

A. DNN Layers

DNNs are composed of layers that implement certain functions. For example, Fig. 2 shows the computation performed

[1]When the precision is required, we will refer to retaining entire intermediate fmaps on-chip as *untiled fusion*, and fusing with inter-layer tiling as *tiled fusion*.

[2]In figures, we will show input fmaps in blue, filters in green, output fmaps in red, intermediate fmaps in purple, and operations in grayscale.

TABLE I
COMPARISON OF DESIGN SPACE IN PRIOR WORKS AND THIS WORK. THIS WORK IS THE FIRST TO SUPPORT EXTENSIVE TILING, RECOMPUTATION, AND PER-TENSOR RETAIN CHOICES

Framework	Partitioned ranks	Recomputation	Per-intermediate-fmap recomputation	Per-tensor retention
TANGRAM [19]	Channel	No	No	Yes
DeFiNES [21]	Row, column	All	No	Yes
ConvFusion [22]	Row, column, channel	Limited[3]	No	Yes
Optimus [28]	Row, column	No	No	Yes
SET [29]	Batch	No	No	Yes
FLAT [30]	Batch, Head, Token	No	No	Yes
TileFlow [31]	Any	Limited[2]	No	No
This work	Any	All	Yes	Yes

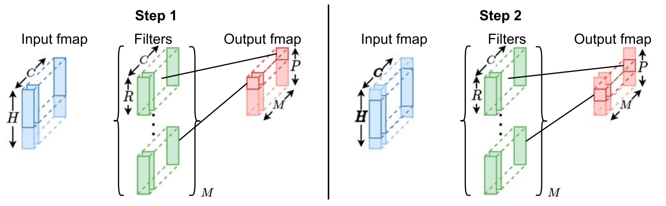

Fig. 2. A 1D conv layer. All input channels (C) are used to generate an output fmap. Values (*i.e.*, activations) in the output fmap column (P) are generated by sliding the convolution window. Different output channels (M) are generated with different filters.

by a 1-dimensional convolution (1D conv) layer. In a 1D conv layer, a window of size R in the input with C channels is multiplied elementwise with a filter of learned weights. The results are accumulated to create a single activation in one channel in the output fmap. The other activations in the output channel are computed by sliding the window and repeating the process. To generate M output channels, M filters are used, one for each output channel. We refer to the values of H, R, C, and M (P can be computed from H and R) collectively as the layer's *shape*. H, R, C, and M themselves are referred to as *ranks*. The computation performed by DNN layers can be described precisely using the *extended Einsum notation*, which we discuss next.

B. The Extended Einsum Notation

The *extended Einsum notation* (hereafter referred to as "Einsum") precisely describes the computation in a layer. First, we discuss the Einsum notation. Then, we discuss how the Einsum notation helps systematize our design space.

To illustrate the Einsum notation, we look again at the 1D convolution in Fig. 2. We can express this layer with a mathematical expression.

$$\text{Output}[m,p] = \sum_{c=0,r=0}^{c=C-1,r=R-1} \text{Input}[c,p+r] \times \text{Filter}[m,c,r] \tag{1}$$

where $m \in [0, M), p \in [0, P)$.

The Einsum notation captures the same computation as Eq. 1 succinctly by allowing implicit inference of the attributes of the summation. In the Einsum notation, Output, Input, and Filter are *tensors* (*i.e.*, multi-dimensional arrays). Each tensor has named *ranks* (written in uppercase letters) and indices that iterate along ranks (written in lowercase). The shape of each tensor is specified by writing the shape of each rank (*i.e.*, the range of legal index values) after an equal sign in the superscript[4] (Table II lists commonly used ranks). Eq. 2 shows the Einsum equivalent to Eq. 1 if we also provide the numerical shape of each rank.

TABLE II
COMMONLY USED RANKS IN CONVOLUTIONAL AND TRANSFORMER LAYERS AND THEIR DEFINITIONS

Ranks in convolutional layers		**Ranks in transformer layers**	
Rank	Definition	Rank	Definition
B	Batches	B	Batches
P	Output height (rows)	H	Heads
Q	Output width (columns)	M	Tokens
M	Output channels	E	Output embedding
C	Input channels	D	Input embedding
H	Input height (rows)		
W	Input width (columns)		
R	Kernel height		
S	Kernel width		

$$\text{Output}_{m,p}^{M=4,P=6} = \text{Input}_{c,p+r}^{C=3,H=8}\text{Filter}_{m,c,r}^{M=4,C=3,R=3} \tag{2}$$

Note that we have explicitly named the input height rank H in Eq. 2 for completeness, whereas Eq. 1 does not. Frequently, the superscripts are easily inferred and omitted for brevity.

Now, we discuss how the Einsum notation helps us describe tiling systematically. The tiling of a tensor can be described as the partitioning of ranks (*e.g.*, splitting the $M = 4$ output channels into $M_1 = 2$ tiles with $M_0 = 2$ channels each). Most prior works assume a preset selection of ranks that can be partitioned (see Table I). As we discuss later, in our design space the user specifies layers as Einsums, and any of the ranks can be partitioned.

Partitioning different ranks results in tiles of operations that reuse tiles of Input, Output, and Filter (*i.e.*, tiles of the data) differently (see Table III). We can arrive at the data reuse pattern by inspecting the Einsum. For example, say we partition rank P in the Einsum in Eq. 2 (see first row in Table III). Then, we define tiles in Input, Output, and Filter to be the subsets

[3]Only a subset of recomputation choices are supported.

[4]This notation is extended from the original Einsum [32] by Hedge et al. [23] to allow affine expressions in the index. Writing the shape of tensors in the superscript is our further extension.

TABLE III
COMPARISON OF DATA REUSE PATTERNS IN INPUT FMAP, OUTPUT FMAP, AND FILTER TENSORS WHEN PARTITIONING DIFFERENT RANKS IN A CNN. "CONV." REFERS TO CONVOLUTIONAL REUSE WHERE TILES PARTIALLY OVERLAP, AS OPPOSED TO A "FULL" REUSE WHERE TILES OVERLAP FULLY

Partitioned ranks	Data reuse in		
	Input	Output	Filter
Row, column (P, Q)	Conv.	None	Full
Input channel (C)	Full	Full	None
Tokens (M)	None	None	Full

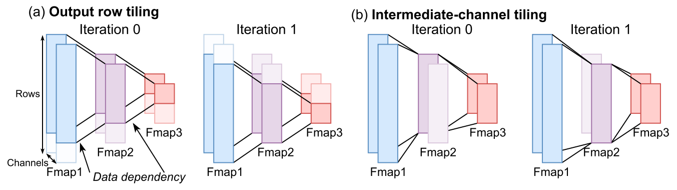

Fig. 3. Examples of different tiling choices. (a) Tiles (darker shade) of Fmap1, Fmap2, and Fmap3 in iterations 0 and 1 in an output row ($P2$ rank) tiling. Note that Fmap2 tiles overlap using this tiling. (b) Tiles (darker shade) of Fmap1, Fmap2, and Fmap3 in iterations 0 and 1 in an intermediate-channel ($C2$ rank) tiling. Note that Fmap2 tiles do *not* overlap using this tiling.

of the tensors accessed by the operation tiles. When data tiles overlap, the data in the overlapping regions are reused. The rank P appears as part of an affine expression $p + r$ in Input, thus partitioning P results in Input tiles that form sliding windows. The activations in the partially overlapping regions are reused in a pattern we refer to as convolutional reuse. Because P appears on its own in Output, the Output tiles do not overlap (*i.e.*, no reuse). Finally, because P does not appear in Filter, partitioning P results in Filter tiles that are the entirety of Filter (*i.e.*, full reuse).

C. Tiling and Retention-Recomputation Choices in Fused-Layer Dataflows

In this section, we provide a brief background on tiling, recomputation, and their interaction in fused-layer dataflows.

Fused-layer dataflows tile layers in order to retain only tiles of intermediate fmaps, thus reducing required on-chip buffer capacity, while still reusing the intermediate fmaps. Fig. 3(a) shows an example of tiling by partitioning the output row of the second layer (*i.e.*, rank $P2$, as in rank P of layer 2).

Fused-layer dataflows can also reduce required on-chip buffer capacity through recomputation. For example, note that tiles in iterations 0 and 1 in Fig. 3(a) overlap. The overlap contains activations in Fmap2 that are computed and used in iterations 0 and 1. At iteration 0, we have two choices: (1) retain the common activations in a buffer to reuse in iteration 1, or (2) do not retain them to save buffer capacity but recompute them later. This *retention-recomputation* choice trades off the buffer capacity required for the intermediate fmap tile with extra computation.

Note that the tiling choice determines the space of retention-recomputation choices. For example, in Fig. 3(b), tiles of Fmap2

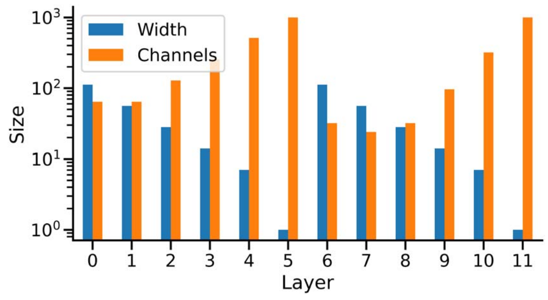

Fig. 4. The width (height is the same as width) and channels of layers in ResNet-18 [34] (layers 1-5) and MobileNetv2 [1] (layers 6-11) vary by orders of magnitude.

are created by partitioning channels of Fmap2. Because the tiles of Fmap2 in different iterations do not overlap, this tiling choice results in no retention-recomputation choice. This relationship between tiling and retention-recomputation choices can only be analyzed by exploring them in combination.

D. Limitations in Prior Fused-Layer Dataflows

Prior fused-layer dataflows and design space exploration (DSE) frameworks have explored only a limited subset of the fused-layer dataflow design space, leaving more efficient designs unexplored (see Table I). We list these limitations and discuss why addressing them leads to more efficient designs.

Limitation 1: tiling choices. As shown in prior work, tiling is needed to efficiently exploit reuse [11], [29], [30], [31], [33]. Tiling is done by partitioning ranks and, as shown in Table III, the choice of partitioned rank impacts the reuse of all tensors. Furthermore, the shape of the layer, which is diverse in DNNs (see Fig. 4), determines the amount of reuse. Combined, this might make a "Full" reuse of a smaller fmap less than a partial "Conv." reuse of a larger fmap. In Section VI, we show that different tiling choices may lead to 10× larger required buffer capacity and that no single tiling choice is universally optimal for every layer shape. However, among prior work, only TileFlow [31] supports an extensive set of partitioned rank choices.

Limitation 2: support for recomputation. As we discussed in Section II-C, fused-layer dataflows can trade off buffer capacity with recomputation. We also discussed that the tiling choice impacts recomputation choices and these choices need to be explored in combination. In Section VI, we show that recomputation may reduce buffer capacity by 2× while adding only 10% more computation. However, prior models either support a limited choice of partitioned ranks or do not support recomputation (see Table I).

Limitation 3: support for per-intermediate-fmap recomputation choices. Conceptually, retention-recomputation choices can be made per-intermediate-fmap. As we will discuss in Section VI-E, being able to make the choice per-intermediate-fmap matters. However, no prior work supports per-intermediate-fmap recomputation choices, and all intermediate fmaps must use the same choice.

Limitation 4: support for per-tensor retention choices. As discussed before, given a particular tiling, the tensors in

TABLE IV
MAPPING CHOICES IN LOOPTREE

Mapping choices	Space of choices
Partitioned ranks	A subset of ranks from the last layer
Tile shape	An integer for each partitioned rank
Tile processing schedule	A permutation of the partitioned ranks
Retain-recompute	One of the partitioned ranks for each intermediate fmap
Retain-refetch	One of the partitioned ranks for each non-intermediate-fmap tensor
Parallelism	"Sequential" or "Pipeline"

our layers may be reused differently (see Table III). It has been shown that making retain choices per tensor, adapting to the particular reuse of each tensor, can lead to more efficient dataflows [27]. In Section VI, we show that per-tensor retention may reduce buffer capacity by 10×. However, none of the prior work that addresses limitations 1 and 2 supports per-tensor retain choices.

III. A More Extensive and Systematic Fused-Layer Dataflow Design Space

In this section, we discuss a design space that addresses the limitations in Section II-D. Formally, we express the design space by describing *mappings*, which is the way operations and data are tiled and scheduled to buffers and compute units [11], [33]. In describing the mappings below, we assume that the user has defined a set of layers to fuse, referred to as a *fusion set*, and an architecture expressed as a set of buffers and compute units. Methods for finding the optimal fusion sets have been explored extensively in prior work (see Section VII).

To create a mapping, the user makes several mapping choices (or, "choices" for short). We list these choices in Table IV. In this paper, we focus on the subset of choices specific to the design of fused-layer dataflows, which we refer to as *inter-layer* choices (see Table IV). Most of the following subsections explain these inter-layer choices. However, we also need to describe how the tile of each layer is processed by specifying the *intra-layer* mapping choices. Intra-layer mapping choices are not the focus of this paper, but poor intra-layer mapping choices will negatively impact the latency and energy of the accelerator [27], [33], [35], [36]. Thus, LoopTree supports them. We discuss intra-layer mapping choices in Section III-E.

A. Tiling by Partitioning Ranks

Fused-layer dataflows employ inter-layer tiling to reduce the buffer capacity required to exploit inter-layer reuse. In LoopTree, we define tiles of the operation space (*i.e.*, the set of operations) of the last layer in the fusion set by partitioning its ranks. Tiles of the data can be determined from the operation tiles via data dependencies (see Fig. 5). Furthermore, we only need to specify the tiling of the last layer because the tiling of all the data and operations of earlier layers can be inferred through data dependencies (see Section IV-A for more detail).

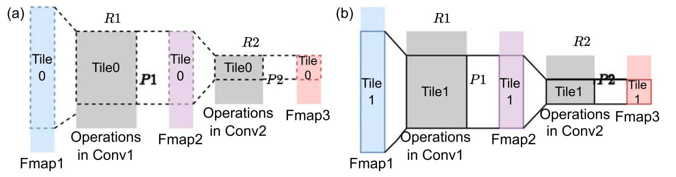

Fig. 5. Partitioning rank $P2$ in Conv2 to create two Conv2 tiles, Tile0 and Tile1. (a) Given the the operation Conv2 Tile0, other data and operation tiles can be inferred. (b) The same is true for Tile1.

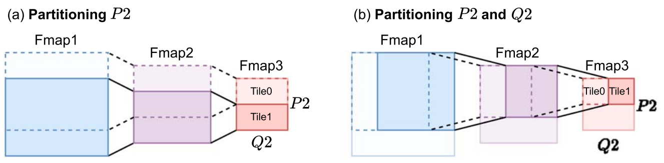

Fig. 6. Examples of partitioning ranks in a 2D conv. (a) Partitioning just $P2$. (b) Partitioning $P2$ and $Q2$.

In general, we can partition any number of ranks to define our tiles (Fig. 6 shows an example). We can also partition the same rank multiple times, which may be useful in architectures with multiple buffer levels. For example, we can partition $P2$ to create tiles such that it fits in a buffer, then partition $P2$ again to get smaller tiles to fit in a lower-level buffer. Note that we have only discussed how to define tiles, while Section III-D discusses how to specify which tiles are retained in buffers.

To finish defining our tiles, we specify the shape of the tile, which is the length of the tile along every rank. The user only needs to specify the length of the tile along partitioned ranks because the tile is assumed to extend to the full length of unpartitioned ranks (*e.g.*, in Fig. 5, only the shape of the tile along $P2$ needs to be specified because the tile extends fully along unpartitioned ranks, such as $R2$). The shape of the tile is commonly chosen such that the tile fits in a buffer.

B. Tile Processing Schedule

After defining the operation tiles, we can specify the scheduling of the operation tiles, which is important because it determines the order in which data is accessed and thus how long we must retain data to achieve a certain amount of reuse. Generally, retaining data for longer requires larger buffers.

Our scheduling follows the constraint that the output from one layer is immediately consumed by the next layer. *E.g.*, in Fig. 5, if we schedule Conv1 Tile0 before Conv1 Tile1, then we produce Fmap2 Tile0 before Fmap2 Tile1 (as in Fig. 5(a)). Thus, we schedule the operations in Conv2 Tile0, which consumes Fmap2 Tile0, before Conv2 Tile1, which consumes Fmap2 Tile1.

In LoopTree, we schedule tile processing by specifying the ranks we want to iterate over, similar to writing loops in a loop nest from the outermost to the innermost. For example, if we follow the $P2, Q2$ schedule to process the tiles in Fig. 6, we will process Tile0, Tile1, then the row underneath. In other words, we iterate across $Q2$ (within a row) first before iterating across $P2$ (across rows). (This schedule is also known as the

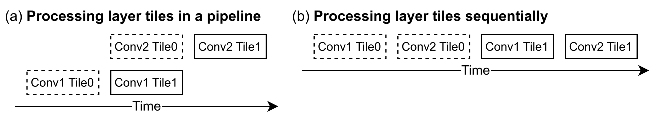

Fig. 7. Parallelism choices: (a) Scheduling tiles across layers in a pipeline (*i.e.*, in parallel); and (b) Scheduling tiles across layers sequentially.

raster scan pattern.) We can also specify $Q2, P2$ where we iterate across $P2$ (within a column) first before going across $Q2$ (across columns).

C. Parallelism

Within each layer, we process tiles in the order we have specified in Section III-B. The scheduling of operation tiles in different layers also needs to follow data dependencies (*e.g.*, in Fig. 5, Conv2 Tile0 has to be processed after Conv1 Tile0). However, we have the freedom to choose the relative timing of subsequent tiles of different layers (*e.g.*, in Fig. 5, Conv1 Tile1 and Conv2 Tile0). We can arrange these tiles to be processed sequentially or in a pipeline (see Fig. 7).

The parallelism choice does not address the limitations of prior work mentioned in Section II-D. However, we support it to be comprehensive such that we can model prior accelerators (see Section V).

D. Data Retention

To achieve on-chip reuse of data, we need to specify which tiles to retain in on-chip buffers and for how long. This choice will determine the required on-chip buffer capacity and the amount of reuse. Here, we will discuss how retention choices are specified in LoopTree, assuming a two-level memory hierarchy of on-chip and off-chip buffers to simplify our discussion.

However, we first observe that although prior work has proposed recomputation as a separate design choice [16], [21], recomputation can be seen as a consequence of our processing schedule and retention choice. Specifically, if we specify a processing schedule and a retention choice such that an operation needs to access an intermediate fmap activation that is not retained in on-chip nor off-chip buffers, then we have to recompute that activation. Note the resemblance with non-intermediate fmap tensors where data that is not retained in on-chip buffers need to be *refetched* from off-chip buffers (note that non-intermediate fmap tensors need to be backed in off-chip buffers). This observation allows us to simplify the design space without restricting the design space: we can specify retention-recomputation choices of intermediate fmaps and retention-refetch choices of non-intermediate fmap tensors using the same representation, which we discuss next.

Section III-A discussed how ranks are partitioned to create operation tiles and data tiles are calculated from operation tiles using data dependencies. In LoopTree, we make a retention choice for each tensor by choosing the last rank partitioned to form the retained tile, which can be one or none of the partitioned ranks, and a buffer in the architecture that retains the data. To illustrate, consider again the tiles formed by

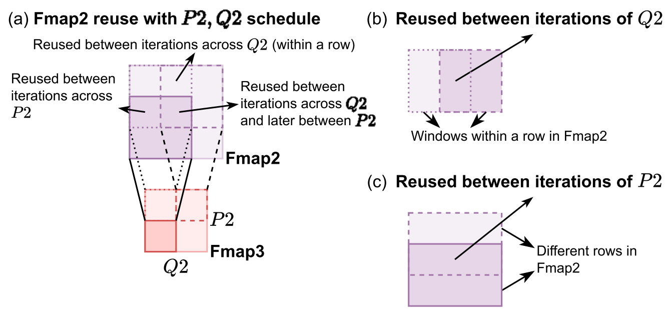
(a) Fmap2 reuse with $P2, Q2$ schedule
Reused between iterations across $Q2$ (within a row)
Reused between iterations across $P2$
Reused between iterations across $Q2$ and later between $P2$
Fmap2
$P2$
Fmap3
$Q2$
(b) Reused between iterations of $Q2$
Windows within a row in Fmap2
(c) Reused between iterations of $P2$
Different rows in Fmap2

Fig. 8. Categorizing data reuse across iterations. (a) Data reuse in Fmap2 tiles with $P2, Q2$ tiling. The reuse can be categorized into two: (b) reuse across iterations of $Q2$ and (c) reuse across iterations of $P2$.

partitioning $P2$ and $Q2$ in Fig. 6. We can retain the entirety of Fmap2, a tile of Fmap2 formed by partitioning the $P2$ rank, or a tile of Fmap2 formed by partitioning the $P2$ and $Q2$ ranks.

Generally, larger tiles result in more reuse, and we illustrate this fact in terms of our retention choice. Fig. 8(a) visualizes the data reuse that results from applying $P2, Q2$ schedule to the tiles in Fig. 6. There are two levels of iterations: across the $P2$ rank (rows) and the $Q2$ rank (columns). Note that the iterations are hierarchical: within a single iteration of the $P2$ rank, we iterate over the entire $Q2$ rank. Fig. 8(b) and 8(c) shows the data reuse categorized by iterations. If we retain Fmap2 tiles formed by partitioning $P2$, we will have data reuse between iterations of $P2$ and $Q2$. We have reuse between iterations of $P2$ because the overlap in data tiles in Fig. 8(c) can be reused between iterations. We have reuse between iterations of $Q2$ because those iterations happen within a single $P2$ iteration and all the data needed will be retained in the buffer. If we retain Fmap2 tiles formed by partitioning $P2$ and $Q2$, we will only have data reuse between iterations of $Q2$. Between iterations of $P2$, the data in the overlapping region in Fig. 8(c) is not guaranteed to still be in the buffer.

Finally, we note that the retained intermediate fmap tile needs to be larger or equal to the intermediate fmap tile produced between layers. Formally, this means that retention choices for intermediate fmaps can only be among the ranks partitioned for the inter-layer tiling. On the other hand, retention choices for non-intermediate-fmap tensors can be made from any of the partitioned ranks, including ranks partitioned in the intra-layer mapping, which we discuss next.

E. Intra-Layer Mapping

At this point, we have a set of operation and data tiles for each layer. All that is left is to specify the order in which we process the elements inside each tile (*i.e.*, specify the intra-layer mapping). We do not go into the details of the intra-layer mapping choices because they are not the focus of this paper and have been explored extensively in prior work [27], [33], [35], [36], [37]. Here, we simply mention that LoopTree supports an extensive set of features from prior work:

- intra-layer tiling for each layer independently,
- mapping to multiple levels of memory hierarchy [27], [33], [35], [36], [37],

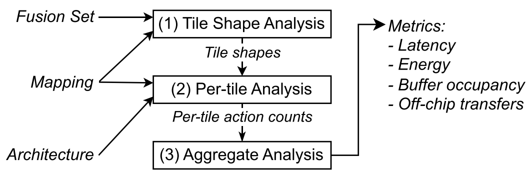

Fig. 9. Overview of the LoopTree model.

- whether to process the tiles sequentially or in parallel [27], [33], [35], [36], [37],
- tiles that are imperfectly factorized [37],
- per-tensor retention (also referred to as uneven mapping) mapping [27].

IV. LoopTree: A Flexible Model for a More Extensive Fused-layer Dataflow Design Space

To explore our *mapspace* (*i.e.*, space of mappings), we need a hardware model for evaluation. Furthermore, we would like this model to be fast and accurate. In this section, we describe a model for our fused-layer dataflow design space. This model leverages patterns in tile shapes and regularity in the hardware behavior when processing identical tile shapes to be able to compute hardware metrics (*e.g.*, latency, energy, buffer occupancy) using mathematical expressions (*i.e.*, the model is *analytical*). Compared to simulators, analytical models tend to be faster at the cost of some fidelity [36].

We briefly overview the analysis steps of the LoopTree model before discussing each step in detail (points match analysis steps in Fig. 9).

1) Given the mapping, which only specifies the tile shape of the last layer, this step determines the operation and data tile shapes of all other layers in the fusion set.
2) Given the tile shapes, architecture, and intra-layer mapping, this step counts the number of occurrences of hardware actions (*e.g.*, buffer reads) during the processing of each tile.
3) Given action counts for each processed tile, this step determines the final metrics (latency, energy, buffer occupancy, and off-chip transfers).

A. Tile Shape Analysis

The user-defined mapping only specifies the tile shapes of the last layer in the fusion set. The tile shape analysis calculates the tile shapes of all the other layers given the fusion set and the mapping. We describe the analysis steps below (see Fig. 10).

Step 1) The mapping specifies the operation tiles of the last layer in the fusion set (Conv3 in Fig. 10).

Step 2) From the operation tile, we can compute the data tile of the input fmap of the last layer (Fmap3 in Fig. 10).

Step 3) From the data tile of the fmap, we subtract the amount that is retained from previous iterations (in Fig. 10, only a subset of the Fmap3 tile needs to be computed).

Step 4) The part of the fmap tile that is not retained from previous iterations has to be produced (in Fig. 10, we calculate the operations of Conv2 required to produce it). This may include the recomputation of certain operations.

Step 5) At this point, we are in the same setup as in step 1, but for an earlier layer (in Fig. 10, Conv2 instead of Conv3). We repeat the analysis until we have calculated the tiles for all the layers in the fusion set.

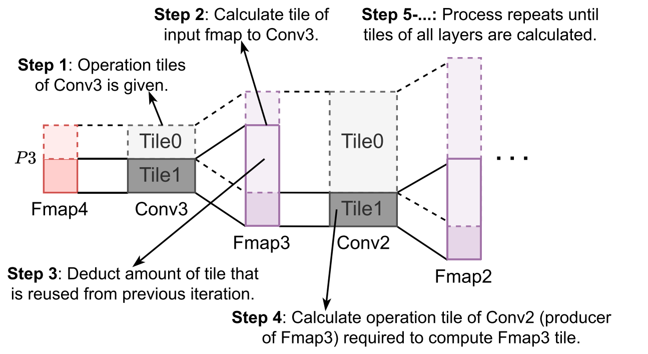
Step 2: Calculate tile of input fmap to Conv3.
Step 5-...: Process repeats until tiles of all layers are calculated.
Step 1: Operation tiles of Conv3 is given.
P3
Tile0
Tile1
Fmap4
Conv3
Fmap3
Tile0
Tile1
Conv2
Fmap2
...
Step 3: Deduct amount of tile that is reused from previous iteration.
Step 4: Calculate operation tile of Conv2 (producer of Fmap3) required to compute Fmap3 tile.

Fig. 10. Analyzing tile shape given inter-layer tiling specification. Layers are shown from last to first (in contrast with other figures) to better illustrate the analysis steps.

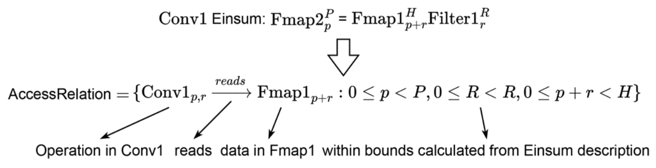
Conv1 Einsum: $\mathrm{Fmap2}_p^P = \mathrm{Fmap1}_{p+r}^H \mathrm{Filter1}_r^R$
AccessRelation $= \{\mathrm{Conv1}_{p,r} \xrightarrow{reads} \mathrm{Fmap1}_{p+r} : 0 \le p < P, 0 \le R < R, 0 \le p+r < H\}$
Operation in Conv1
reads
data in Fmap1
within bounds calculated from Einsum description

Fig. 11. Calculating the data access relation as a polyhedron from the Einsum description. Polyhedrons have efficient implementations for set operations (*e.g.*, intersection, union, etc.).

In the steps we just described, we needed to calculate data tiles from operation tiles and data dependencies (*e.g.*, in step 1) and vice versa (*e.g.*, in step 3). We also needed to perform set operations on operation tiles (*e.g.*, in step 2). We briefly discuss a fast implementation of these calculations. As mentioned at the start of this section, our approach to evaluation is *analytical*. Rather than explicitly constructing operation tiles and the data accesses (*i.e.*, which data each operation accesses), we represent them as a set constrained by equalities and inequalities containing affine expressions (see Fig. 11). In other words, we construct polyhedral sets and relations [38]. There are fast methods for performing these set and relation operations [39]. While storing sets and relations explicitly requires storage proportional to the size of the DNN we are modeling, which can be enormous, our analytical approach requires storage proportional to the number of tiling steps and the number of ranks in the fusion set. The same is true for the number of operations required in our subroutines. This allows the LoopTree model to be fast (prior work has shown analytical models to be up to $1000\times$ faster than simulators [36]).

B. Per-Tile Hardware Action Counts Analysis

Given the inter-layer tiles of each layer, architecture, and intra-layer mappings, we now have to analyze the hardware actions (*e.g.*, reads, writes, computes) required when processing each tile. Because this analysis is performed for each layer independently, we can use a similar analysis to layer-by-layer frameworks [27], [33], [35], [36], [37] (the analysis in LoopTree is based on Timeloop [33]). However, we implement this analysis using set and relation operations so that it is compatible with the results of the tile shape analysis[5]. While this implementation is not trivial, it is not the main focus of this paper. Thus, we refer the readers to the open-source documentation and mention here several notable features of the analysis.

- LoopTree analyzes the impact of temporal data reuse on the number of transfers between buffer levels in the memory hierarchy. For example, if the same data is required within a buffer between two iterations, LoopTree detects this reuse opportunity and the data is not refetched from the parent buffer.
- We analyze the impact of spatial data reuse (*i.e.*, when the same data is needed by multiple compute units) on buffer reads and data transfers. For example, if two child buffers need to receive the same data at the same time, LoopTree detects the multicast opportunity and counts only a single read to the parent buffer. Furthermore, LoopTree will calculate the number of hops required to send the data from the parent to the child buffers on the network-on-chip (NoC).

The result of this analysis is a set of hardware action counts accumulated during the processing of the tiles. Specifically, we have:

- Reads and writes to each buffer in the hardware.
- The number of network hops to distribute data in each network in the hardware.
- The latency required to process all the operations at the compute units.

Finally, we note that the processing of tiles with the same shape will have the same behavior and therefore the same hardware counts. Our model detects unique tile shapes and performs the analysis only once for each unique tile shape.

C. Analyzing Final Metrics

In the final analysis, we use the action counts from the processing of each tile to compute the final metrics.

1) Calculating Total Latency: The latency calculation is divided into two cases: (1) if the tiles are processed sequentially and (2) if the tiles are processed in a pipeline. In the sequential case, latency can be calculated as the sum of the latencies of the processing of each tile.

The pipeline case is more complicated. In Fig. 10, the number of operations in Tile0 and Tile1 Conv2 is different because the reused part of Fmap3 does not have to be recomputed. This makes the number of operations in a tile depend on which iteration the tile belongs to. LoopTree's algorithm for calculating total latency takes this into account.

LoopTree's algorithm for evaluating pipeline latency first arranges the pipeline stages sequentially (*i.e.*, with no pipeline parallelism) (see Fig. 12(a) and 12(b)). Overlapping the stages hides some latency. In Fig. 12(b), they are HiddenLatency1, HiddenLatency2, and HiddenLatency3. The actual hidden latency is the minimum of the three. Then, the total pipeline latency is the sequential latency minus the hidden latency.

[5]An alternative is to provide an adapter. However, we hope that the polyhedral implementation will make future extensions easier.

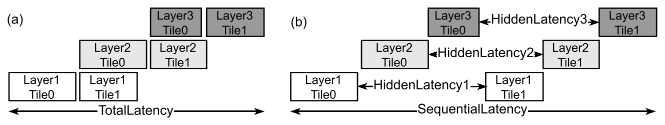

Fig. 12. Pipeline latency analysis. (a) Total latency of the processing of three layers in a pipeline. (b) Latency of sequential processing of the layer tiles and the latencies that could be hidden in a pipeline.

Similarly to other analyses in LoopTree, the analysis avoids duplicate work by using a polyhedral representation of tiles.

Finally, the latency we computed so far is only the computation latency. We aggregate read/write counts to each buffer and divide by the bandwidth to get memory latency. We take the larger one of the compute and memory latencies to be the final latency. We note that this analysis assumes that data layout reordering is performed during processing such that the reordering does not increase latency. There are existing state-of-the-art works demonstrating the feasibility of such processing, *e.g.*, the *reorder in reduction* technique introduced by Tong et al. [40]. Moreover, LoopTree assumes explicit data orchestration using *Buffets* [41] such that pipeline stalls can be assumed to be negligible.

2) Calculating Total Energy: Given the action counts (*i.e.*, read/write counts, the number of computations, network hops, and peer-to-peer transfers) of intra-layer processing (see Section IV-B) and inter-layer processing (see Section IV-A), we can calculate the energy consumption of the system by multiplying the counts of each action with the energy per action. The energy per action is generated from the architecture specification using Accelergy [42].

3) Calculating Buffer Occupancy: Given a design, we have to make sure that the buffer occupancy does not exceed the buffer capacity. From the tile shape inference, we know the data that has to occupy each buffer. We can then compute the buffer occupancy from the shape of the tiles. The sequential processing of each layer tile may require different amounts of buffer occupancy. LoopTree reports the maximum buffer occupancy.

4) Calculating Off-Chip Transfers: As mentioned in Subsection IV-B, off-chip transfers (which is a special case of buffer-to-buffer data transfers) are computed per-tile in the last step. The total off-chip transfers are the sum of the per-tile off-chip transfers. LoopTree can also report other buffer-to-buffer transfers.

V. Validation

We implement and validate the accuracy of the LoopTree analytical model across prior architectures that cover a wide range of the design choices described in Section III and DNNs with varying reuse patterns.

TABLE V
HIGH-LEVEL SUMMARY OF VALIDATIONS. DNN LAYERS: CONVOLUTION (CONV), POINTWISE CONVOLUTION (PWISE), DEPTHWISE CONVOLUTION (DWISE), SELF-ATTENTION (SA). PARALLELISM: SEQUENTIAL (S), PARALLEL (P). OUTPUTS: LATENCY (L), ENERGY (E), CAPACITY (C), OFF-CHIP TRANSFERS (T)

Design	DNN type	DNN layers	Partitioned ranks	Retain-recompute	Parallelism	Output				Max. error %
						L	E	C	T	
DepFin [43]	CNN [44], [45]	conv, dwise, pwise	Row, column	Fully retain	s		✓	✓	✓	0
Fused-layer CNN [16]	CNN [3], [4]	conv	Row, column	Fully retain	p	✓		✓	✓	1.2
ISAAC [17]	CNN [3]	conv	Column	Fully retain	p		✓	✓		4
PipeLayer [18]	CNN [3], [4]	conv	Batch	Fully retain	p	✓				3.3
FLAT [30]	Transformers [6]	sa	Heads, tokens, batch	Fully retain	s	✓		✓	✓	3.4

A. Methodology

We implement LoopTree in C++ as an extension of Timeloop [33]. We implement the analysis described in Section IV, using the ISL library [39] to handle set and relation operations. LoopTree uses Accelergy [42] as the energy estimation back end.

B. Validation Setup

For validation, we choose designs that exercise a wide range of LoopTree's capabilities. Table V summarizes these designs. We choose five fused-layer dataflow accelerators for convolutional neural networks (CNNs) and transformers. They are chosen because they span different fused-layer dataflow design choices across a range of DNNs. For each design, we model each design executing the same DNNs as the ones they were evaluated on in the publication. Then, based on available information, we compare latency, energy consumption, buffer capacity, and off-chip transfers.

C. Validation Results

Overall, LoopTree shows less than 4% error. Here, we discuss each validation result in Table V in detail.

1) DepFin: DepFin [43] is a CNN accelerator that partitions P and Q (because the number of layers in the fusion set differs in each validation, we omit the number in the rank name that represents the layer in this section) and processes tiles sequentially with a P, Q schedule. The DNN models are FSRCNN [45], MC-CNN [44], and the two in-house CNN models in the publication [43]. These DNNs contain vanilla convolutions as well as pointwise and depthwise convolutions, which have different reuse patterns. LoopTree results exactly match the energy and off-chip transfer counts as reported in the paper. LoopTree also validates the result that DepFin has enough buffer capacity to retain data given its fusion set, mapping, and architecture.

2) Fused-Layer CNN: Fused-layer CNN [16] is a CNN accelerator that partitions P and Q and processes tiles in a pipeline with a P, Q schedule. Because the paper only reports the number of BRAMs used, we create a simulation based on the architecture description for the buffer capacities. Table VI shows LoopTree's results, the simulator result, and the result after synthesis on an FPGA reported in the paper. All errors are within 1.2%. Energy was not reported in [16].

TABLE VI
FUSED-LAYER CNN RESULTS. TABLE SHOWS LOOPTREE, REFERENCE, AND SYNTHESIZED RESULT. REFERENCE RESULT IS SIMULATED BASED ON PSEUDOCODE OF THE ARCHITECTURE IN [16]. LOOPTREE RESULTS ARE WITHIN 1.2%

Metric	LoopTree	Ref.	Synth.
Latency (kcycles)	422		427
WBuf capacity (KB)	167	167	
IOBuf capacity (KB)	44	44	
TBuf capacity (KB)	6	6	
Off-chip transfers (KB)	611	611	688

TABLE VII
ISAAC BUFFER CAPACITY REQUIREMENT. REFERENCE IS THE RESULTS IN [17]. LOOPTREE MATCHES REFERENCE RESULTS

DNN	LoopTree (KB)	Ref. (KB)
VGG-1-conv1	1.96	1.96
VGG-1-conv2	21	21
VGG-1-conv3	21	21
VGG-1-conv5	21	21

The slight error in latency is possibly due to implementation details in the synthesized design. The error in off-chip transfers could be caused by LoopTree's assumption of ideal data layout in memory. In practice, off-chip memory block sizes might not match the tile shapes exactly.

3) ISAAC: ISAAC [17] is a CNN accelerator that partitions Q and processes tiles in a pipeline. We model their dataflow in LoopTree using the same assumption of balanced throughput. Table VII shows the buffer capacities calculated by LoopTree and the reported values. Furthermore, we model their energy consumption in LoopTree. LoopTree recreates their reported energy efficiency with at most 4% error. The difference could be attributed to a slight mismatch between LoopTree's action-based modeling and ISAAC's power-throughput model.

4) PipeLayer: PipeLayer [18] is a CNN accelerator that partitions B and processes tiles sequentially or in a pipeline. PipeLayer shows speedup when processing tiles in a pipeline. We model both their sequential and pipelined dataflow in LoopTree. LoopTree replicates the latency results in [18] with

TABLE VIII
PIPELAYER SPEEDUP DUE TO PIPELINING. REFERENCE FROM [18]. LOOPTREE RESULTS ARE WITHIN 3.3% OF REFERENCE

DNN	LoopTree	Ref.
AlexNet	4.8	4.8
VGG-A	7.9	8.0
MNIST-A	2.0	2.0
MNIST-B	2.9	3.0

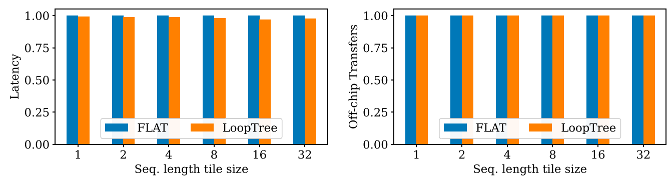

Fig. 13. Normalized (a) latency and (b) off-chip transfers generated by the FLAT [30] simulator and LoopTree (this work). Results differ by at most 3.4%.

3.3% error (see Table VIII). The difference could be due to unmodeled aspects of the ReRAM arrays (*e.g.*, latency from loading weights).

5) FLAT: FLAT [30] is an accelerator for transformers [8] that partitions B, H, and M and processes tiles sequentially with a B, H, M schedule. We modeled the FLAT dataflow with different tile shapes using LoopTree. Fig. 13 shows the normalized results. In all experiments, LoopTree results differ by at most 3.4%. The most divergent results are latency. The small difference stems from aspects in the FLAT simulator that LoopTree does not model (*e.g.*, latency from loading weights and systolic array startup).

VI. CASE STUDIES

We discuss a series of case studies that illustrate the trade-off between on-chip buffer capacity, recomputation, and off-chip transfers in fused-layer dataflow design. Specifically, we evaluate the impact of design aspects in Table I (formalized in Section III) and how they interact.

Table IX summarizes the setup for each case study. Each case study investigates the impact of design space choices by setting those choices as independent variables and searching for other choices that lead to the optimal in a certain metric. Specifically, in Section VI-B, we evaluate the impact of (inter-layer) partitioned ranks and tile processing schedule (or, "schedule" for short) without recomputation. In Section VI-C, we allow recomputation and evaluate the impact of partitioned ranks and schedule choices on the trade-off between recomputation and on-chip buffer capacity. In Section VI-E, we look at the impact of being able to make recomputation choices per-intermediate-fmap. Then, in Section VI-D, we show how per-tensor retention leads to a smaller required on-chip buffer capacity. Finally, in Section VI-C, we show how LoopTree can be used to evaluate the overall impact of tiled fusion.

A. Introducing the Fusion Sets

Before discussing the case studies, we discuss the three fusion sets we will use. Table X shows the Einsums of the fusion sets, which tells us the distinctive features of each fusion set:

1) Two 2D convolutions (conv) layers. These are the 2D convolutional layers we have discussed so far.
2) Three layers consisting of pointwise (pwise), depthwise (dwise), and pointwise (pwise) convolutions. A notable difference from conv layers is that pwise layers do not have convolutional reuse (*e.g.*, Fmap3 in Table X have $p3$ instead of $p3 + r3$ index) and dwise layers do not have channel reuse (in Table X, the $M2$ rank is shared by Fmap2, Fmap3, and Filter2).
3) Two fully connected layers (fc) modeled after those in transformers [6], [7], [8]. This fusion set only has full reuse and no reuse; there is no convolutional reuse.

In the Einsums, we constrain the fusion set shapes to match common shapes in recent DNNs. Table X shows the shape of the layers in the fusion sets. (For conciseness, we will refer to the "shapes of layers in a fusion set" collectively as the *fusion set shape*.) The bolded rank names are variables we will change to vary the fusion set shapes.

B. Impact of Partitioned Ranks and Schedule Choices

In this case study, we evaluate the impact of partitioned ranks and schedule choices on the required on-chip buffer capacity to achieve algorithmic minimum off-chip transfers (*i.e.*, the minimum achievable off-chip transfers assuming infinite on-chip buffer capacity) without recomputation. To simplify the following discussion, we refer to only the schedule choice and imply the partitioned ranks. For example, a $P2, C2$ schedule implies that we create tiles by partitioning $P2$ and $C2$, and the tiles are processed with a $P2, C2$ schedule. Fig. 14, shows the on-chip buffer capacity required by the optimal partitioned ranks and schedule choices (which change with fusion set shape) and two other choices for comparison.

Fig. 14 shows that partitioned ranks and schedule choices have a significant impact on the required on-chip buffer capacity (*e.g.*, the capacity required by a $P2$ and $C2$ schedule may differ by up to 10×). The reason is that the partitioned ranks and schedule determine which tensor needs to be stored on-chip. For example, with the $P2$ schedule in conv+conv, we fully reuse Filter1 and Filter2 because they are required to process every operation tile. Thus, if we do not want to refetch Filter1 and Filter2 multiple times from off-chip buffers, we must keep those tensors on-chip. Then, in fusion sets where Filter1 and Filter2 are large (*e.g.*, when there are many channels), Filter1 and Filter2 significantly increase the required on-chip buffer capacity.

As a corollary of the above, the optimal choice tends to correspond with the largest rank. For example, when there are many rows (*i.e.*, $P2$ is large), the $P2$ schedule results in a smaller required on-chip buffer capacity. This is because, in fusion sets where the $P2$ rank is large compared to other ranks, tensors with the $P2$ rank tend to be larger than other tensors (*e.g.*, the size of Fmap1 is proportional to $P2$, but Filter1 is not).

TABLE IX
OVERVIEW OF CASE STUDY SETUP. THE "CASE STUDY" COLUMN REFERS TO THE SUBSECTION THAT DISCUSSES THE CASE STUDY

Case study	Partitioned Rank	Tile Shape	Schedule	Retention (for non-intermediate tensors)	Retention-recomputation (for intermediate fmaps)
B	Independent	Searched	Independent	Searched	Searched s.t. no recomputation
C	Independent	Searched	Independent	Searched	Searched
D	Searched	Searched	Searched	Independent	Searched
E	Searched	Searched	Searched	Searched	Independent
F	Searched	Searched	Searched	Searched	Searched s.t. no recomputation

TABLE X
LAYERS COMPRISING THE FUSION SETS WE WILL USE TO EVALUATE FUSED-LAYER DATAFLOWS. THE RANK SHAPES COLUMN SHOWS WHICH RANKS HAVE EQUAL VALUES. BOLDED TEXT DENOTES RANK VALUES WHICH WE WILL VARY THROUGHOUT THE EXPERIMENT TO GET FUSION SETS OF DIFFERENT SHAPES

Fusion set	Einsums	Fusion set shapes	Modeled after
conv+conv	$\text{Fmap2}^{M1,P1,Q1}_{m1,p1,q1} = \text{Fmap1}^{M1,P1,Q1}_{c1,p1+r1,q1+s1}\text{Filter1}^{C1,M1,R1=3,S1=3}_{c1,m1,r1,s1}$ $\text{Fmap3}^{M2,P2,Q2}_{m2,p2,q2} = \text{Fmap2}^{C2,P2,Q2}_{c2,p2+r2,q2+s2}\text{Filter2}^{C2,M2,R2,S2}_{c2,m2,r2,s2}$	**Rows**: $P1 = Q1 = P2 = Q2$ **Channel**: $C1 = M1 = C2 = M2$	ResNet blocks [34]
pwise+dwise+pwise	$\text{Fmap2}^{M1,P1,Q1}_{m1,p1,q1} = \text{Fmap1}^{C1,P1,Q1}_{c1,p1,q1}\text{Filter1}^{C1,M1}_{c1,m1}$ $\text{Fmap3}^{M2,P2,Q2}_{m2,p2,q2} = \text{Fmap2}^{M2,P2,Q2}_{m2,p2+r2,q2+s2}\text{Filter2}^{M2,R2=3,S2=3}_{m2,r2,s2}$ $\text{Fmap4}^{M3,P3,Q3}_{m3,p3,q3} = \text{Fmap3}^{C3,P3,Q3}_{c3,p3,q3}\text{Filter2}^{C3,M3}_{c3,m3}$	**Rows**: $P1 = Q1 = P2 = Q2$ **Channel**: $C1 = M3$ $C1 = \frac{M1}{6} = \frac{M2}{6} = \frac{C3}{6}$	MobileNetv2 blocks [1]
fc+fc	$\text{Fmap2}^{M1,E1}_{m1,e1} = \text{Fmap1}^{M1,D1}_{m1,d1}\text{Filter1}^{D1,E1}_{d1,e1}$ $\text{Fmap3}^{M2,E2}_{m2,e2} = \text{Fmap2}^{M2,D2}_{m2,d2}\text{Filter2}^{M2,E2}_{m2,e2}$	**Tokens**: $M1 = M2$ **Emb. dims.**: $E1 = D2$ $D1 = E2 = 1024$	Transformer feed-forward blocks [8]

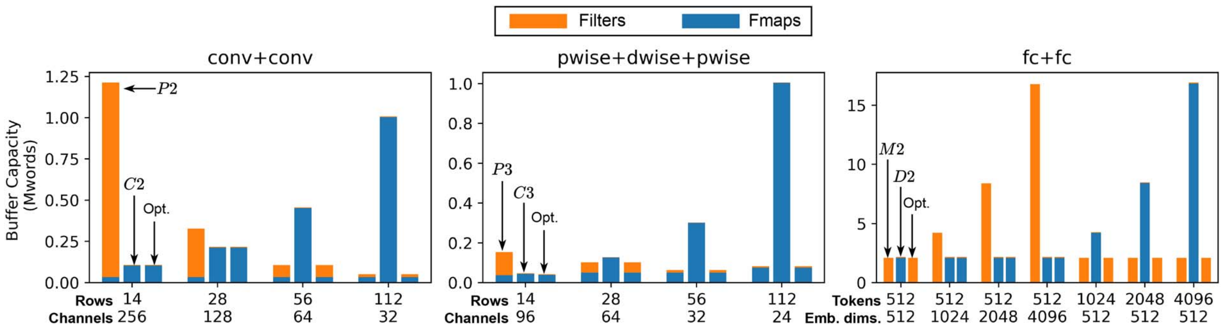

Fig. 14. Buffer capacity required for alg. min. off-chip transfers using different partitioned rank and schedule choices. Subplots: different fusion sets. Groups of bars: different fusion set shapes (see Table X col. 3) Bars: different partitioned ranks and schedules. We show three bars: the optimal (opt.) choice and two other choices for comparison. Partitioned ranks and schedule choices have a significant impact on buffer capacity across different types of fusion sets and fusion set shapes.

Thus, partitioning $P2$ means we are tiling the larger tensors and fully reusing the smaller ones. Because we have to store the entirety of the fully reused tensors, this leads to a smaller on-chip buffer capacity.

The pwise+dwise+pwise fusion set in Fig. 14 shows a similar trend in buffer capacities and breakdowns as the conv+conv fusion set. However, the total filter size is generally smaller, which is a feature of the MobileNet design [1]. As a result, the $P3$ schedule, which is better when filters are smaller than fmaps, is the optimal choice for more fusion set shapes.

The reasoning can also be applied to fully-connected layers. For example, using the $M2$ schedule results in fully reusing (and thus retaining the entirety of) the filters. On the other hand, using the $D2$ schedule results in retaining the entirety of Fmap1 and Fmap3. Thus, we see the pattern in Fig. 14.

Takeaway 1: the partitioned ranks and schedule that results in the smallest required on-chip buffer capacity is often the one that reuses the smallest tensors.

C. Impact of Partitioned Ranks and Schedule Choices With Recomputation

In this case study, we evaluate the impact of partitioned ranks and schedule choices on the required on-chip buffer capacity to achieve algorithmic minimum off-chip transfers with recomputation. We compare the Pareto front of required

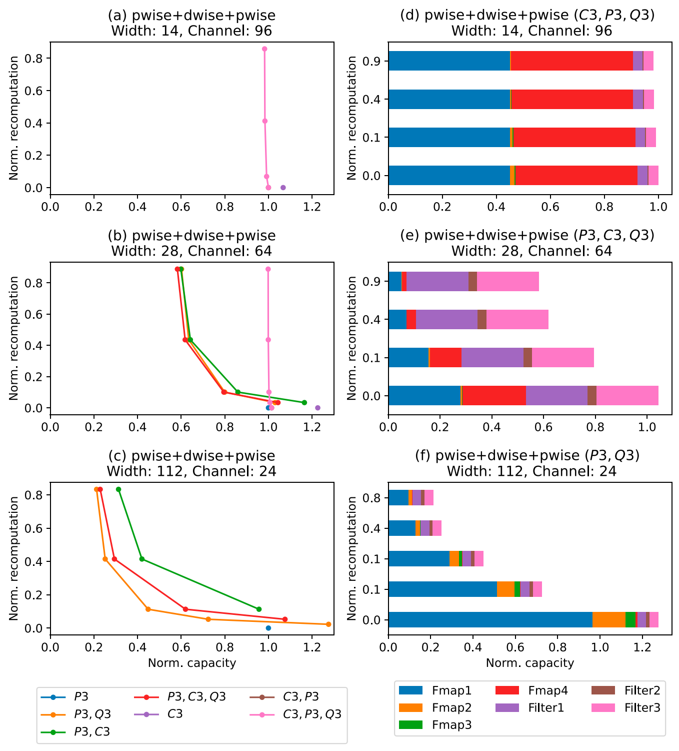

Fig. 15. (a)–(c) Normalized recomputation against the normalized capacity for different partitioned ranks and schedule (colors) and fusion set shapes (subgraphs). The optimal partitioned ranks and schedule choice vary by fusion set shape. (d)–(f) Breakdown of buffer capacity usage by tensors for the partitioned ranks and schedule choice in parentheses.

on-chip buffer capacity and recomputation (*i.e.*, the set of mappings that achieve the fewest recomputations and smallest required on-chip buffers) for different partitioned ranks and schedule choices. Fig. 15 shows the Pareto front for pwise+dwise+pwise. We make three observations.

First, recomputation changes which partitioned ranks and schedule result in a smaller required on-chip buffer capacity. For example, Fig. 15(b) and 15(c) show that the partitioned ranks and schedule that results in the smallest required on-chip buffer capacity without recomputation is $P3$, which is different with recomputation ($P3, C3, Q3$ in Fig. 15(b) and $P3, Q3$ in Fig. 15(c)). Thus, retention-recomputation, partitioned ranks, and schedule choices need to be explored together.

Second, the partitioned ranks and schedule that results in the most efficient recomputation and capacity trade-off differs for different fusion set shapes (*e.g.*, it is $P3, C3, Q3$ in Fig. 15(b) and $P3, Q3$ in Fig. 15(c)). Thus, it is important to search from an extensive set of partitioned rank choices because no particular choice results in the smallest required on-chip buffer capacity at a given recomputation amount for all fusion set shapes.

The reason behind the second observation is similar to the one discussed in the last subsection. The partitioned ranks and schedule choice determine which tensors are fully reused and thus fully retained. In Fig. 15(a), there are many channels, but the width is small, thus the filters are larger than fmaps. Any of the partitioned ranks and schedule that starts with $P3$ needs to

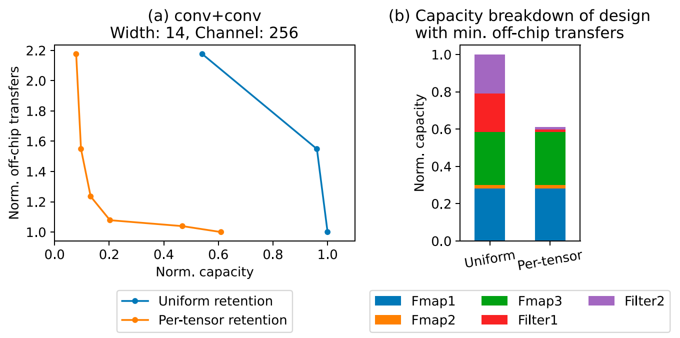

Fig. 16. (a) Normalized off-chip transfers against normalized buffer capacity with uniform and per-tensor retention. Using per-tensor retention can reduce buffer capacity by up to 40%. (b) Capacity breakdown of the design with min. off-chip transfers for uniform and per-tensor retention.

fully retain the filters, which significantly increases the required on-chip buffer capacity. Comparing Fig. 15(a)–15(c), the trend reverses as fmaps become larger than filters.

Third, note that the slope of the Pareto frontier differs for different partitioned ranks and schedules. When the slope is steep (*e.g.*, $C3, P3, Q3$ in Fig. 15(a)), more recomputation does not lead to significant required on-chip buffer capacity reduction. The breakdown of the capacity usage (see Fig. 15(d)) explains the steep slope: with the $C3, P3, Q3$ schedule, Fmap1 and Fmap4 need to be fully retained. We can reduce buffer capacity by retaining smaller tiles of Fmap2 and Fmap3 at the cost of more recomputation. However, because the Fmap2 and Fmap3 tiles are only small portions of the required capacity, the reduction in required capacity is insignificant. Compare this with $P3, Q3$ in Fig. 15(c) and 15(f). With $P3, Q3$, the filters need to be fully retained. However, the fmaps still take up the majority of the on-chip buffer capacity. Thus, retaining smaller tiles of the fmaps results in significant capacity reduction.

Finally, we note that the fc+fc fusion set does not have retention-recomputation choices because all partitioned ranks and schedule choices for fc+fc result in intermediate fmap tiles that do not overlap.

Takeaway 2: retention-recomputation, partitioned ranks, and schedule choices need to be explored together, and it is important to search from an extensive set of partitioned rank choices because no particular choice results in the smallest buffer capacity for all fusion set shapes.

D. Impact of Per-Tensor Retention Choices

We evaluate the impact of having per-tensor retention choices compared to uniform retention choices for the conv+conv fusion set (other fusion sets show similar results). In this evaluation, we do not consider recomputation. Because the LoopTree mapspace allows per-tensor retention, we explore the default mapspace to evaluate mappings with per-tensor retention. To evaluate mappings with uniform retention, we constrain the mapspace such that the same retention choice is made for all tensors. Fig. 16(a) shows normalized off-chip transfers against normalized on-chip buffer capacity for the conv+conv fusion set with per-tensor retain and uniform retain.

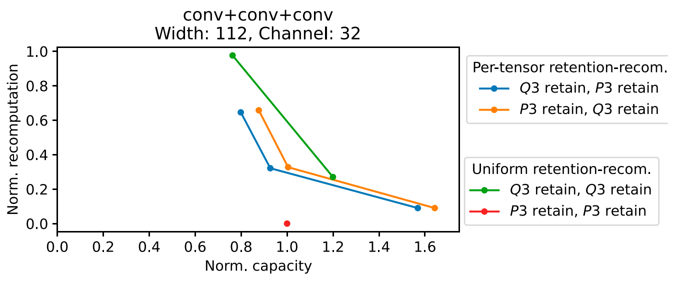

Fig. 17. Normalized buffer capacity and normalized recomputation Pareto curves for $P2, Q2$ for different retain-recompute choices. The legend lists retain-recompute choices for Fmap2 and Fmap3 respectively. Mixing retain-recompute choices for different fmaps leads to better tradeoffs than a uniform retain-recompute choice.

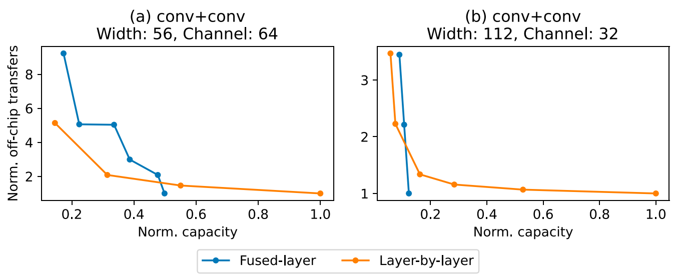

Fig. 18. Buffer capacity and off-chip transfers tradeoff Pareto curves for tiled fused-layer and baseline without recomputation. At capacity lower than required for minimum algorithmic off-chip transfers, the baseline is often better than fused-layer.

Fig. 16(a) shows that per-tensor retention can reduce required on-chip buffer capacity significantly (up to $9\times$). To illustrate why, we pick the per-tensor and uniform retention mappings with the fewest off-chip transfers (the lowest point of each curve) and show the capacity usage breakdown in Fig. 16(b). Filter1 and Filter2 require much smaller on-chip buffer capacity with per-tensor rather than with uniform retention. For a given schedule, different tensors have different minimum tile shapes that need to be retained to avoid refetches. Making per-tensor retention choices allows us to match these minimum tile shapes. Here, the uniform retention choice retains larger filter tiles than necessary to avoid refetches due to its constraints.

Takeaway 3: Per-tensor retention choices result in a smaller required on-chip buffer capacity because we can adapt the retention choice to the reuse pattern of each tensor.

E. *Impact of Per-Tensor Retention-Recomputation Choices*

When there are multiple intermediate fmaps, LoopTree allows us to make retention-recomputation choices per tensor. Here, we evaluate the impact of per-tensor retention-recomputation choices on a fusion set of three convolutional layers (conv+conv+conv)[6] for the $P3, Q3$ schedule. There are two intermediate fmaps in the fusion set (Fmap2 and Fmap3) and for each intermediate fmap, we can either choose to retain across $P3$ or $Q3$. Without support for per-tensor choices, we can only make the same choice for all the tensors (*i.e.*, uniform choices). By being able to make per-tensor retention-recomputation choices, we can also make different choices in addition to uniform choices. Fig. 17 shows that per-tensor choices result in a smaller required on-chip buffer capacity for the same recomputation amount. Furthermore, recomputing Fmap2 and retaining Fmap3 leads to a smaller required on-chip buffer capacity than recomputing Fmap3 and retaining Fmap2.

To understand the result, note that recomputing Fmap3 requires more activations in Fmap2 to be retained as inputs. If we are retaining Fmap2, then a larger on-chip buffer is required to retain Fmap2. If we are recomputing Fmap2, then the amount of Fmap2 recomputation increases compared to if we retained Fmap3 (*i.e.*, whether we choose to retain or recompute Fmap3 impacts the number of recomputations of Fmap2). Thus, recomputing later layers increases the required on-chip buffer capacity or recomputations in earlier layers. Prior work referred to this as a compounding effect in recomputation [16].

Takeaway 4: per-tensor retention-recomputation choices can reduce the amount of recomputation by limiting the compounding of recomputations.

F. *Overall Impact of Tiled Fusion*

In this case study, we compare tiled fused-layer designs against a baseline that picks the best of either processes layer-by-layer or retains entire intermediate fmaps (*i.e.*, untiled fusion). We take the conv+conv fusion set with shapes where $P2, Q2$ is the optimal choice. Then, we evaluate the required on-chip buffer capacity required for different amounts of off-chip transfers without recomputation. Fig. 18 shows the results.

In Fig. 18, we can see that the number of off-chip transfers rises faster for the tiled fused-layer dataflow as available on-chip buffer capacity is reduced. As a result, the baseline dataflow is more efficient when on-chip buffers are small.

To see why, we make two observations:

- Intermediate fmap activations are reused between layers twice: one read and write saved for every activation.
- Intra-layer reuse is often more abundant. For example, in the conv+conv fusion set with 64 channels, each activation in the fmap is read by $3 \times 3 \times 64 = 576$ operations (activations near the borders are reused fewer times, but these are few).

Thus, given limited capacity, it is more efficient to exploit the more abundant intra-layer reuse.

On the other hand, achieving algorithmically minimum accesses using tiled fused-layer dataflow requires a significantly smaller capacity, compared to the baseline which has to retain the entire intermediate fmap to achieve minimum accesses. This result shows the advantage of inter-layer tiling in achieving inter-layer reuse efficiently.

Of course, there exist factors that may cause fused-layer dataflows to be more efficient than layer-by-layer even if the available on-chip capacity is low: certain workloads may have

[6]This fusion set is omitted in Table X because it is used only in this case study. We use three conv layers such that we have two intermediate fmaps that have retention-recomputation choices. We do not use pwise+dwise+pwise because Fmap3 does not have retention-recomputation choices.

few intra-layer reuse opportunities (*e.g.*, in elementwise operations) or if compute units are especially efficient, which may make recomputation cheaper. As we have demonstrated, LoopTree can be used to explore these trade-offs to find efficient designs.

Takeaway 5: at buffer capacities much lower than required for algorithmic minimum off-chip transfers, fused-layer dataflows are often less efficient than layer-by-layer dataflows.

VII. Related Works

A. Fused-Layer Dataflow Accelerators

Prior work has proposed fused-layer dataflow accelerators (under various names) [16], [17], [18], [19], [30], [43], [46]. These works have shown that fused-layer dataflow accelerators can have lower latency, lower energy consumption, or higher scalability compared to layer-by-layer dataflow accelerators. This makes the fused-layer dataflow accelerator design space interesting to explore. However, although these works often describe a hardware performance model and an algorithm to search for an optimal mapping, each model and mapping search approach are specific to the proposed accelerator and the target workload. Thus, a more general model that supports a wide fused-layer dataflow accelerator design space and various workloads is needed.

B. Fused-Layer Dataflow Accelerator Exploration Framework

Fused-layer dataflow exploration frameworks (*e.g.*, [21], [22], [28], [29], [31] often support a range of fused-layer dataflow accelerators (instead of a specific one) and workloads. However, as discussed in Sections I and II, prior frameworks have limitations in certain features of their design spaces. This work addresses the aforementioned limitations. Furthermore, although each framework may have only a subset of the limitations, our case studies in Section VI have shown that exploring these features in combination results in more efficient accelerator designs.

Prior work has also proposed methods for determining optimal fusion sets [22], [28], [29], [46]. For example, Optimus [28] is a method based on dynamic programming, SET [29] is based on simulated annealing, ConvFusion [22] is based on Pareto-filtering. Orthogonally, LoopTree is a model to find the optimal design choices for a fusion set. Thus, LoopTree can be used in conjunction with any of these methods.

C. Search Algorithms for Design Space Exploration

Prior work has explored using various search algorithms to explore the mapspace [22], [28], [29], [46], [47], [48], [49]. Although not necessarily exploring the mapspace of fused-layer accelerators, many of these search algorithms can be adapted to search the LoopTree mapspace using LoopTree as the model, *e.g.*, dynamic programming [28], genetic algorithms [49], or differentiable surrogate in [48].

VIII. Conclusion

In this paper, we described a fused-layer mapspace with more extensive tiling, recomputation, and retention choices than prior work. We also described and validated an analytical hardware model that can evaluate latency, energy, and required buffer capacity given a mapping and fusion set. Using this model, we explore the fused-layer mapspace for fusion sets in CNNs and transformers, revealing insights into fused-layer dataflow accelerator design that can only be explored with a model supporting a more extensive fused-layer dataflow design space.

References

[1] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobileNetV2: Inverted residuals and linear bottlenecks,” in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 4510–4520.

[2] A. Howard et al., “Searching for MobileNetV3,” in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, 2019, pp. 1314–1324.

[3] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *Proc. Int. Conf. Learn. Representations*, 2015.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 25, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Red Hook, NY, USA: Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[5] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, “Natural language processing (almost) from scratch,” *J. Mach. Learn. Res.*, vol. 12, no. null, pp. 2493–2537, Nov. 2011.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” Oct. 2018, *arXiv:1810.04805*.

[7] G. Lample and A. Conneau, “Cross-lingual language model pretraining,” Jan. 2019, *arXiv:1901.07291*.

[8] A. Vaswani et al., “Attention is all you need,” in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Red Hook, NY, USA: Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[9] A. Hannun et al., “Deep speech: Scaling up end-to-end speech recognition,” 2014, *arXiv:1412.5567*.

[10] M. Bansal, A. Krizhevsky, and A. Ogale, “ChauffeurNet: Learning to drive by imitating the best and synthesizing the worst,” Dec. 2018, *arXiv:1812.03079*.

[11] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, 2016, pp. 367–379.

[12] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” in *Proc. IEEE Int. Solid-State Circuits Conf., (ISSCC), Dig. Tech. Papers*, 2016, pp. 262–263.

[13] H. Kwon, A. Samajdar, and T. Krishna, “MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects,” *SIGPLAN Notes*, vol. 53, no. 2, pp. 461–475, Mar. 2018, doi: 10.1145/3296957.3173176.

[14] T. Chen et al., “DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” *SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 269–284, Feb. 2014, doi: 10.1145/2654822.2541967.

[15] “NVIDIA deep learning accelerator,” [Online]. Available: http://nvdla.org/primer.html

[16] M. Alwani, H. Chen, M. Ferdman, and P. Milder, “Fused-layer CNN accelerators,” in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2016, pp. 1–12.

[17] A. Shafiee et al., “ISAAC: A convolutional neural network accelerator with *in-situ* analog arithmetic in crossbars,” in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, 2016, pp. 14–26.

[18] L. Song, X. Qian, H. Li, and Y. Chen, “PipeLayer: A pipelined ReRAM-based accelerator for deep learning,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2017, pp. 541–552.

[19] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, “TANGRAM: Optimized coarse-grained dataflow for scalable NN accelerators,” in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst., (ASPLOS)*, New York, NY, USA: ACM, 2019, pp. 807–820, doi: 10.1145/3297858.3304014.

[20] N. P. Jouppi et al., “Ten lessons from three generations shaped Google’s TPUv4i : Industrial product,” in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2021, pp. 1–14.

[21] L. Mei, K. Goetschalckx, A. Symons, and M. Verhelst, “DeFiNES: Enabling fast exploration of the depth-first scheduling space for DNN accelerators through analytical modeling,” in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA),* Los Alamitos, CA, USA: IEEE Comput. Soc. Press, Mar. 2023, pp. 570–583, doi: 10.1109/HPCA56546.2023.10071098.

[22] L. Waeijen, S. Sioutas, M. Peemen, M. Lindwer, and H. Corporaal, “ConvFusion: A model for layer fusion in convolutional neural networks,” *IEEE Access*, vol. 9, pp. 168245–168267, 2021.

[23] K. Hegde et al., “ExTensor: An accelerator for sparse tensor algebra,” in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, (*MICRO*), New York, NY, USA: ACM, 2019, pp. 319–333, doi: 10.1145/3352460.3358275.

[24] T. O. Odemuyiwa, J. S. Emer, and J. D. Owens, “The EDGE language: Extended general einsums for graph algorithms,” Apr. 2024, *arXiv:2404.11591*.

[25] N. Nayak, X. Wu, T. O. Odemuyiwa, M. Pellauer, J. S. Emer, and C. W. Fletcher, “FuseMax: Leveraging extended Einsums to optimize attention accelerator design,” in *Proc. 57th Annu. IEEE/ACM Int. Symp. Microarchit., (MICRO)*, New York, NY, USA: ACM, 2024.

[26] N. Nayak, T. O. Odemuyiwa, S. Ugare, C. W. Fletcher, M. Pellauer, and J. S. Emer, “TeAAL: A declarative framework for modeling sparse tensor accelerators,” Apr. 2023, *arXiv:2304.07931*.

[27] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst, “ZigZag: Enlarging joint architecture-mapping design space exploration for DNN accelerators,” *IEEE Trans. Comput.*, vol. 70, no. 8, pp. 1160–1174, Aug. 2021.

[28] X. Cai, Y. Wang, and L. Zhang, “Optimus: An operator fusion framework for deep neural networks,” *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 1, pp. 1–26, Oct. 2022, doi: 10.1145/3520142.

[29] J. Cai, Y. Wei, Z. Wu, S. Peng, and K. Ma, “Inter-layer scheduling space definition and exploration for tiled accelerators,” in *Proc. 50th Annu. Int. Symp. Comput. Archit., (ISCA),* New York, NY, USA: ACM, 2023, pp. 1–17, doi: 10.1145/3579371.3589048.

[30] S.-C. Kao, S. Subramanian, G. Agrawal, A. Yazdanbakhsh, and T. Krishna, “FLAT: An optimized dataflow for mitigating attention bottlenecks,” 2021, *arXiv:2107.06419*.

[31] S. Zheng et al., “TileFlow: A framework for modeling fusion dataflow via tree-based analysis,” in *Proc. 56th Annu. IEEE/ACM Int. Symp. Microarchit., (MICRO)*, New York, NY, USA: ACM, 2023, pp. 1271–1288, doi: 10.1145/3613424.3623792.

[32] A. Einstein, “The foundation of the general theory of relativity,” *Annalen der Physik*, vol. 354, no. 7, pp. 769–822, 1916.

[33] A. Parashar et al., “Timeloop: A systematic approach to DNN accelerator evaluation,” in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2019, pp. 304–315.

[34] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” Dec. 2015, *arXiv:1512.03385*.

[35] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, “MAESTRO: A data-centric approach to understand reuse, performance, and hardware cost of DNN mappings,” *IEEE Micro*, vol. 40, no. 3, pp. 20–29, May/Jun. 2020.

[36] Y. N. Wu, P.-A. Tsai, A. Parashar, V. Sze, and J. S. Emer, “Sparseloop: An analytical, energy-focused design space exploration methodology for sparse tensor accelerators,” in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2021, pp. 232–234.

[37] M. Horeni, P. Taheri, P.-A. Tsai, A. Parashar, J. Emer, and S. Joshi, “Ruby: Improving hardware efficiency for tensor algebra accelerators through imperfect factorization,” in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2022, pp. 254–266.

[38] W. Pugh, “Uniform techniques for loop optimization,” in *Proc. 5th Int. Conf. Supercomput. (ICS)*, New York, NY, USA: ACM, 1991, pp. 341–352, doi: 10.1145/109025.109108.

[39] S. Verdoolaege, “isl: An integer set library for the polyhedral model,” in *Proc. Math. Softw. (ICMS),* vol. 6327, K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, Eds., Berlin, Heidelberg: Springer, 2010, pp. 299–302.

[40] J. Tong, A. Itagi, P. Chatarasi, and T. Krishna, “FEATHER: A reconfigurable accelerator with data reordering support for low-cost on-chip dataflow switching,” in *Proc. 51th Annu. Int. Symp. Comput. Archit., (ISCA)*, New York, NY, USA: ACM, 2024.

[41] M. Pellauer et al., “Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration,” in *Proc. 24th Int. Conf. Architect. Support Program. Lang. Oper. Syst., (ASPLOS)*, New York, NY, USA: ACM, 2019, pp. 137–151, doi: 10.1145/3297858.3304025.

[42] Y. N. Wu, J. S. Emer, and V. Sze, “Accelergy: An architecture-level energy estimation methodology for accelerator designs,” in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*, 2019, pp. 1–8.

[43] K. Goetschalckx, F. Wu, and M. Verhelst, “DepFiN: A 12-nm depth-first, high-resolution CNN processor for IO-efficient inference,” *IEEE J. Solid-State Circuits*, vol. 58, no. 5, pp. 1425–1435, May 2023.

[44] J. Żbontar and Y. LeCun, “Stereo matching by training a convolutional neural network to compare image patches,” Oct. 2015, *arXiv:1510.05970*.

[45] J. Żbontar and Y. LeCun, “Stereo matching by training a convolutional neural network to compare image patches,” Oct. 2015, *arXiv:1510.05970*.

[46] S. Zheng, X. Zhang, L. Liu, S. Wei, and S. Yin, “Atomic dataflow based graph-level workload orchestration for scalable DNN accelerators,” in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2022, pp. 475–489.

[47] Q. Huang et al., “CoSA: Scheduling by constrained optimization for spatial accelerators,” in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2021, pp. 554–566.

[48] K. Hegde, P. A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, “Mind mappings: Enabling efficient algorithm-accelerator mapping space search,” in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS),* Piscataway, NJ, USA: IEEE Press, 2021.

[49] S.-C. Kao and T. Krishna, “GAMMA: Automating the HW mapping of DNN models on accelerators via genetic algorithm,” in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des. (ICCAD)*, 2020, pp. 1–9.