TileFlow: A Framework for Modeling Fusion Dataflow via Tree-based Analysis

Size Zheng
Peking University
China
zhengsz@pku.edu.cn

Siyuan Chen
Peking University
China
chensiyuan@pku.edu.cn

Siyuan Gao
Peking University
China
2000012960@stu.pku.edu.cn

Liancheng Jia
Peking University
China
jlc@pku.edu.cn

Guangyu Sun
Peking University &
Advanced Innovation Center for
Integrated Circuits
China
gsun@pku.edu.cn

Runsheng Wang
Peking University &
Advanced Innovation Center for
Integrated Circuits
China
wrs@pku.edu.cn

Yun Liang*
Peking University &
Advanced Innovation Center for
Integrated Circuits
China
ericlyun@pku.edu.cn

ABSTRACT

With the increasing size of DNN models and the growing discrepancy between compute performance and memory bandwidth, fusing multiple layers together to reduce off-chip memory access has become a popular approach in dataflow design. However, designing such dataflows requires flexible and accurate performance models to facilitate evaluation, architecture analysis, and design space exploration. Unfortunately, current state-of-the-art performance models are limited to the dataflows of single operator acceleration, making them inapplicable to operator fusion dataflows.

In this paper, we propose a framework called TileFlow that models dataflows for operator fusion. We first characterize the design space of fusion dataflows as a 3D space encompassing compute ordering, resource binding, and loop tiling. We then introduce a tile-centric notation to express dataflow designs within this space. Inspired by the tiling structure of fusion dataflows, we present a tree-based approach to analyze two critical performance metrics: data movement volume within the accelerator memory hierarchy and accelerator compute/memory resource usage. Finally, we leverage these metrics to calculate latency and energy consumption. Our evaluation validates TileFlow's modeling accuracy against both real hardware and state-of-the-art performance models. We use TileFlow to aid in fusion dataflow design and analysis, and it helps us discover fusion dataflows that achieve an average runtime speedup of 1.85× for self-attention and 1.28× for convolution chains compared to the state-of-the-art dataflow.

CCS CONCEPTS

• **Computing methodologies** → *Machine learning algorithms*; • **Hardware** → *Power estimation and optimization*; **Analysis and design of emerging devices and systems**.

KEYWORDS

Tensor Programs, Fusion, Accelerator, Simulation and modeling

ACM Reference Format:
Size Zheng, Siyuan Chen, Siyuan Gao, Liancheng Jia, Guangyu Sun, Runsheng Wang, and Yun Liang. 2023. TileFlow: A Framework for Modeling Fusion Dataflow via Tree-based Analysis. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3613424.3623792

1 INTRODUCTION

Deep learning has made incredible strides over the last few years. Different models with various layers have been proposed in the areas of image classification [11, 19, 57, 59, 60], object detection [16, 17, 37, 50, 51] , image generation [27, 52, 83] , and natural language processing [3, 10, 53, 61]. To accelerate the computation in DNN models, various spatial accelerators [7, 12, 13, 23, 36, 39, 41] have been proposed. *Spatial accelerators* employ processing engine (PE) arrays to exploit data reuse and parallelism. Different spatial accelerators use different dataflows for acceleration. *Dataflow* refers to how to schedule data and computation in hardware resources

*Corresponding Author

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3623792

over space and time. For example, Google TPU [23] uses weight-stationary systolic arrays to accelerate matrix multiplication and convolution; Sanger [39] proposes a score-stationary dataflow that keeps sparse scores stationary in PE to accelerate sparse attention layers.

However, as the discrepancy between the speed of on-chip computing units and off-chip memory increases, the overhead of data transfer between on-chip units and off-chip memory becomes a bottleneck to performance [9, 79]. It is no longer sufficient to accelerate one operator at a time. Instead, *fusion dataflow* that stages the intermediate results in fast on-chip buffers to reduce off-chip memory access overhead is becoming prevalent. A fusion dataflow refers to the multi-operator execution plan that stages data spatially and temporally from off-chip memory through on-chip memory hierarchy to compute PEs [26]. And a dataflow for a specific input shape with concrete tiling sizes is often called a *mapping*. For example, Fused-Layer [2] proposes a tile-stationary dataflow to fuse convolution operators by staging the intermediate image tiles in on-chip memory; FLAT [26] designs different dataflows for self-attention layers to stage a block of rows of data in on-chip memory for softmax operator.

Designing fusion dataflows is challenging for three reasons. First, it's difficult to decide the order of memory access and compute operations for different operators. The execution order determines the reuse distance of intermediate data and thus determines the lifetime and available memory resource for each intermediate data. Changing the order of two operators may also influence the execution of other operators, resulting in a complex design space to explore. Second, it's non-trivial to decide hardware resource binding for each operator in a fusion dataflow. There is a trade-off between performance and resource usage. For example, pipelining different layers provides good latency at the cost of high compute and memory resource usage, while sequentially executing each layer can alleviate resource pressure but results in a long latency. Third, a high-performance dataflow requires a careful selection of tiling loops and tiling factors to coordinate the memory access and computation latency under hardware resource constraints. Manually exploring the tiling space is impossible.

To design high-performance dataflows, performance models are crucial. However, state-of-the-art performance models [31, 38, 45] focus on single operator acceleration. These models fail to provide flexible and accurate performance prediction for fusion dataflows. On one hand, these models treat the computation of a single operator as a polyhedron of iterations and formulate the performance prediction problem as calculating a polynomial composed of architecture parameters such as PE array size and workload parameters such as iteration bounds. Therefore, we classify these models as *polyhedron-based*. But the single polyhedron formulation is not suitable for fusion dataflow because the iteration space of a fusion dataflow is not perfectly nested. Fusion will insert the iteration space of one operator into the iteration space of another operator, forming imperfect loop nests. Other works [67, 72] modify existing models to evaluate fusion dataflow performance by first modeling each operator separately using the performance model and then stripping the unneeded inter-operator data movement latency from the results. We call these works *graph-based* because they only consider the compute graph topology in modeling without consideration for architecture details (e.g., memory hierarchy). This approach also requires a great expertise in performance model implementation and is only feasible when input workloads and architecture specifications are known ahead of time, prohibiting the exploration for new workloads or accelerator designs.

In this paper, we propose TileFlow, a framework for modeling fusion dataflows on spatial accelerators. **Our insight** is that fusion dataflow is not a perfect polyhedron, but is a tree structure. So the modeling analysis should be designed for the tree structure as well. To do this, we first clearly characterize the three dimensions of fusion dataflow design space (compute ordering, resource binding, and loop tiling) in TileFlow. We propose to express different dataflows in the 3D design space through a tile-centric notation, which can be converted to an analysis tree for fusion dataflow.

Then, to analyze the fusion dataflow, we propose a *tree-based* analysis approach to uniformly calculate the performance metrics of fusion dataflows for general DNN layers. Data movement volume calculation and resource usage are calculated from bottom to top using the dataflow analysis tree. In detail, the dataflow analysis tree is composed of tile nodes. A tile node represents a polyhedron of iterations over its children nodes and it may also carry binding information about the resource partition/sharing for its children nodes. TileFlow calculates the data movement volume using the tree structure, binding information, and tiling decisions for every single tile (intra-tile) and every pair of tiles (inter-tile).

At last, based on these performance metrics, we can infer the latency and energy with respect to architecture specifications. To help design space exploration, we combine genetic algorithm and Monte Carlo Tree Search to implement a searching algorithm in TileFlow's mapper. To the best of the authors' knowledge, TileFlow is the first framework that can systematically model the performance of fusion dataflows for general DNN layers and customized accelerator architectures. We also make TileFlow open-source in Github (https://github.com/pku-liang/TileFlow). In summary, our contributions are as follows:

- We clearly characterize the complete 3D design space for fusion dataflows on spatial accelerators and provide a tile-centric notation to express dataflow designs in the 3D space.
- We propose a tree-based analysis approach to calculate performance critical metrics: data movement volume and resource usage.
- We implement the proposed modeling techniques into a framework called TileFlow to help dataflow/mapping evaluation, architecture analysis, and design space exploration.

In evaluation, TileFlow's modeling accuracy is validated using both real hardware and state-of-the-art performance model [45]. We use TileFlow to help fusion dataflow design and analysis. TileFlow can find better dataflows that achieve 1.85× average runtime speedup compared to state-of-the-art work [2, 26] for self-attention and 1.28× speedup for convolution chains.

2 BACKGROUND

2.1 Spatial Accelerator Architecture

Spatial accelerators are often designed in a hierarchy. We show a typical spatial accelerator architecture design in Figure 1 part

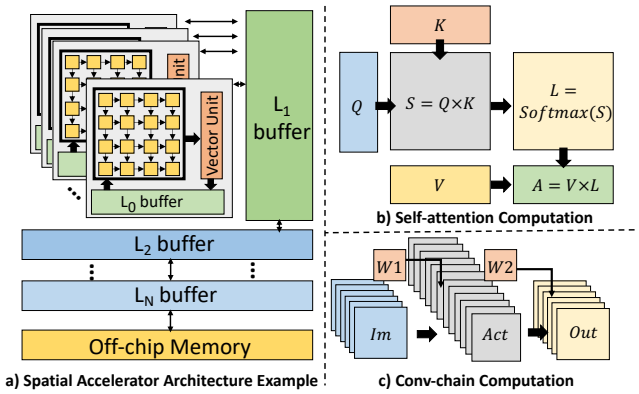


Figure 1: Spatial accelerator architecture and the important workloads in DNN models.

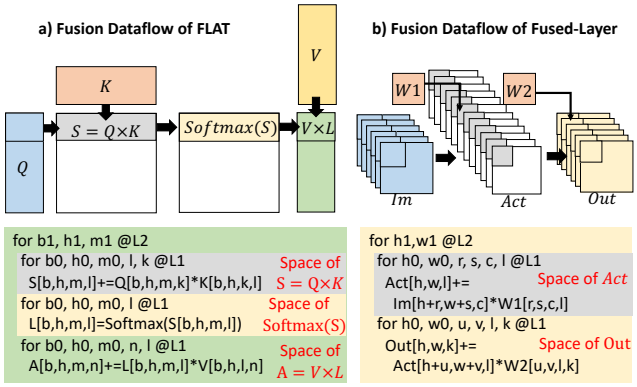


Figure 2: Fusion dataflows of FLAT [26] and Fused-Layer [2].

a). The innermost level is composed of processing engine (PE) arrays that support various computation workloads such as matrix multiplication, convolution, and vector operations at a small scale (e.g., $16 \times 16 \times 16$ matrix multiplication). The PE array has a fast L0 buffer (e.g., register) to stage input/output data. Outer levels are composed of multiple levels of on-chip memory hierarchy (L_1 - L_N buffer) and the outermost level is off-chip memory (e.g., DRAM). Commodity accelerators such as Tensor Core GPU [41], TPU [23], and NPU [36] all use spatial architecture.

2.2 DNN Layers Fusion and Dataflow

As the DNN model size continues to grow, the performance bottleneck switches to memory bandwidth for recent models such as Transformer models [9, 79]. In Figure 1 part b) and part c), we show two important workloads in DNN. In part b), it is a self-attention layer composed of two batch matrix multiplications and one softmax operator ($S = Q \times K, L = Softmax(S), A = V \times L$). The intermediate tensors S, L are large and grow quadratically with input sequence length. In part c), we show a convolution chain composed of two convolutions ($Act = Conv(Im, W1), Out = Conv(Act, W2)$). In many CNNs [55, 73], the intermediate tensor Act can also be larger than both input and output tensors. The intermediate tensor size expansion may make these DNN workloads memory-bound.

To alleviate the bottleneck in memory bandwidth, various fusion dataflows have been proposed to stage intermediate results in on-chip memory and reduce off-chip memory access. A fusion dataflow refers to the execution plan about how to stage data from off-chip memory through the on-chip memory hierarchy to compute PEs [26]. Different dataflows may use different execution order, resource binding, and loop tiling strategies, resulting in different performance (e.g., latency and energy). For example, FLAT [26] proposes a row-based dataflow for self-attention layer that stages a fixed number of rows for the first batch matrix multiplication ($S = Q \times K$) and the softmax operator. We show the dataflow in Figure 2 part a). Fused-Layer [2] proposes a tile-based dataflow for convolution chains by staging a tile of intermediate results (Act) in on-chip memory. We show the dataflow in Figure 2 part b).

2.3 Existing Dataflow Performance Models

Designing an efficient dataflow is challenging. To help dataflow design, various performance models have been proposed. Timeloop [45] uses temporal/spatial notations to describe different dataflows on customized architectures; MAESTRO [31] uses a data-centric notation for dataflow expression and supports various operators in DNN; TENET [38] uses relation-centric notation to provide accurate latency estimation for tensor applications. These performance models focus on single operator acceleration. They abstract the DNN layers as a polyhedron of iterations and treat a dataflow as a series of transformations of the polyhedron over space and time. As a result, performance modeling problem is formulated into the calculation of a set of polynomials composed of architecture parameters (PE size, bandwidth, etc.) and workload parameters (inputs shape, strides, etc.). We classify these performance models as *polyhedron-based*. However, these models fail to model the performance of fusion dataflows because, for a fusion dataflow, the iteration space is not a perfect polyhedron. Fusing one operator (Op_1) into another operator (Op_2) is to insert the iteration space of Op_1 into the iteration space Op_2. So after fusion, the iteration space of the fused workload is not perfectly nested.

Other lines of work [67, 72] handle fusion by first evaluating each operator separately on polyhedron-based models and then eliminate unwanted inter-operator data transfer according to the DNN model topology. We classify these works as *graph-based*. Such an approach requires a lot of expertise in stripping out the unneeded inter-operator data movement, which is error-prone and inflexible as the specific workload and architecture parameters (e.g., bandwidth) should be known ahead of time.

Compared to existing work [4, 14, 33], our TileFlow proposes to use *tree-based* analysis to evaluate fusion dataflows. TileFlow uses a tile-centric notation to express fusion dataflows so that an analysis tree of the dataflow can be captured naturally. TileFlow can express compute ordering, resource binding, and loop tiling (3D space) using the analysis tree. GCNAX [33] and OMEGA [14] are specially designed to express and evaluate fusion dataflows for GNN [30]. They mainly consider loop tiling and ordering for dataflow modeling (2D space). By contrast, TileFlow further includes resource binding into the design space so that the accelerator's on-chip memory hierarchy can be modeled and evaluated.

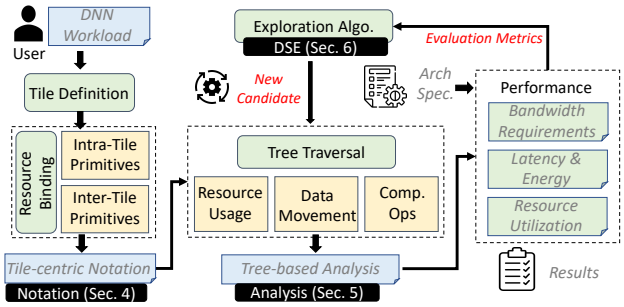


Figure 3: Overview of TileFlow.

SET [4] proposes to use resource allocation tree (RA tree) to express inter-layer fusion dataflows with consideration of loop ordering and resource binding. Then, it decides the loop tiling through intra-layer scheduling. However, SET's design space is limited because it uses DNN layers as the scheduling unit and only allows pipelining among mini-batches. As a result, SET's design space is between 2D (ordering and binding) and 3D (ordering, binding, and tiling). By contrast, TileFlow can express the full 3D space via the tile-centric notation. Each layer is split (at any possible dimension, not limited to mini-batches) into tiles and the scheduling unit is the tile. Therefore, different scheduling possibilities (e.g., pipelining and parallelization) can be exploited among fine-grained tiles.

3 OVERVIEW OF TILEFLOW

The overall workflow of TileFlow is shown in Figure 3. TileFlow is mainly composed of two parts: tile-centric notation and tree-based analysis. The input of TileFlow is a DNN workload provided by the user written in TileFlow's tile-centric notation. We focus on dense workloads in this paper. The tile-centric notation has complete expressiveness in the 3D design space of fusion dataflow. Compute ordering and loop tiling are expressed by the tile definition, while resource binding is expressed by two kinds of primitives: intra-tile primitives and inter-tile primitives. We explain the details in Section 4.

A fusion dataflow expressed by tile-centric notations can be naturally converted to an analysis tree. TileFlow then traverses the analysis tree to calculate performance-critical metrics: data movement volume within the on-chip memory hierarchy, resource usage of each level of memory and compute units, and the total number of computation operations required by the accelerator. With these metrics, performance results including latency, energy, and bandwidth requirements are calculated based on the architecture specifications. We explain the tree-based analysis in Section 5. We also design a mapper in TileFlow to explore the design space of fusion dataflow. We introduce the mapper in Section 6.

4 TILE-CENTRIC NOTATION

4.1 3D Design Space of Fusion Dataflow

We characterize the design space of fusion dataflow as a 3D space composed of three dimensions: compute ordering, resource binding, and loop tiling. Compute ordering is to choose the proper order of execution for all the operators in the given DNN workload. Resource binding is to allocate compute and memory resources for each step of computation and for each operator. Loop tiling is about choosing loops to tile and finding optimized tiling factors to maximize performance with respect to the hardware resource constraints. In Figure 4 part a), we show an example DNN workload with three operators. In part b), we visualize the 3D design space for this workload.

Table 1: The resource binding primitives of TileFlow.

Name	Abbr.	Explanation	Example
Intra-tile Primitives			
Spatial	**Sp**	map loops to spatial units	**Sp**(loops)
Temporal	**Tp**	map loops to temporal steps	**Tp**(loops)
Inter-tile Primitives			
Sequential	**Seq**	tiles each occupies all the hardware resources in turns	**Seq**(T_1, ..., T_M)
Sharing	**Shar**	tiles share the hardware memory and execute in turns	**Shar**(T_1, ..., T_M)
Parallel	**Para**	tiles spatially use different compute and memory units	**Para**(T_1, ..., T_M)
Pipeline	**Pipe**	tiles are dependent and execute in a pipeline manner	**Pipe**(T_1, ... , T_M)

First, we explain the details for the compute ordering dimension. Each design point in this dimension is an *ordering tree*, indicating the execution order of operators. Our insights in choosing tree structure are two folds. On one hand, tree structure is suitable for tiling because outer loops can form the root node, while inner loops can form the children nodes. On the other hand, the tree structure can naturally capture the insertion of one polyhedron into another polyhedron, which corresponds to the fusion of operators. Each node in the tree represents a tile of computation, which is a polyhedron composed of iterations over its children nodes. When an operator Op_1 is fused into another operator Op_2, Op_1's iteration space (a polyhedron) is inserted into the iteration space of Op_2, which is modeled as inserting a node (for Op_1) as the child to another node (for Op_2) in the tree structure. In Figure 4 part c), we zoom in one ordering tree choice and show its structure in detail. This tree structure means that operator A is fused to operator B at $L1$ memory, and both of them are fused to operator C at $L2$ memory (the color of each tile indicates which operator the tile belongs to). The execution order is: in $L1$ memory, compute a tile of A first and then compute a tile of B; repeat the first step until the $L2$ tile of B is ready; after this, use the data tile of B from $L2$ to compute a tile of C in $L2$ memory; finally, repeat the aforementioned steps until C is fully completed.

Besides the ordering of tiles, the loop orders within one tile should also be carefully decided because the loop orders influence the fusion granularity. In detail, when fusing two operators (fuse Op_1 into Op_2), only the reduction loops of Op_2 are allowed in the parent tile in the result ordering tree (as shown in Figure 4 part c). Otherwise, if the reduction loops of Op_1 appear in parent tile (as outer loops), Op_2 can't start execution until Op_1 has finished. As a result, the fusion pipeline is inefficient.

Second, we explain the details for resource binding. Each design point in this dimension is a choice of resource partition or sharing,

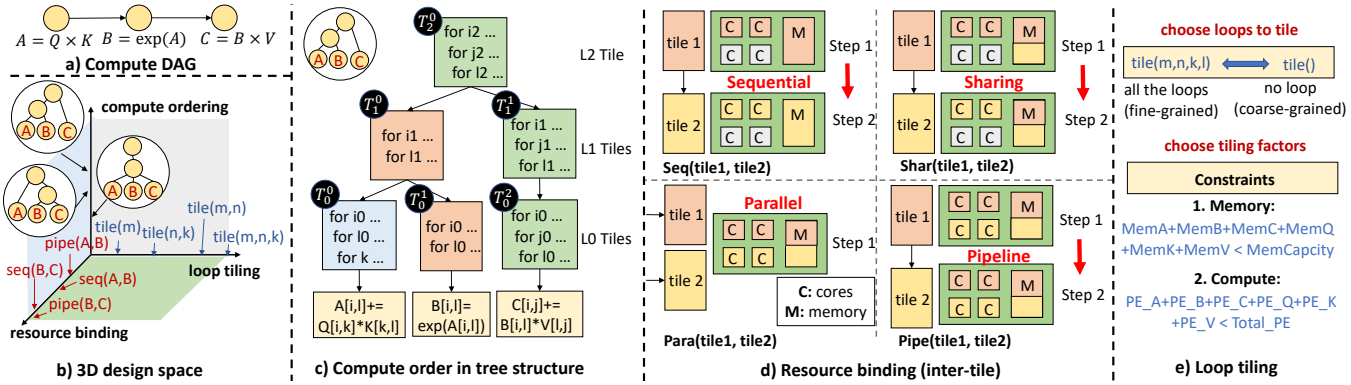


Figure 4: The 3D design space of fusion dataflow.

which is represented as a primitive. For the computation within one node in an ordering tree, we use intra-tile primitives; for the computation from different nodes, we use inter-tile primitives. These two types of primitives are summarized in Table 1. The explanation of *Sp* and *Tp* are well discussed in polyhedron-based models [31, 38, 45]. So we mainly explain the inter-tile primitives. We use Figure 4 part d) to explain the inter-tile primitives. *Seq* binds each tile of computation to the same hardware resources exclusively (without any sharing) in different execution steps. *Shar* binds each tile to the same compute resources in different execution steps but shares the memory resources for them for different steps. *Para* binds each tile to different parts of compute and memory resources in the same time step, enabling spatial sharing of hardware. *Pipe* also spatially shares hardware resources, but the tiles are executed in a pipeline manner.

There is a complex trade-off in latency and resource usage for these primitives. Depending on the input shapes, different primitives are suitable for different tiles. *Pipe* can reduce latency and improve throughput at the cost of high compute and memory resource usage. But for large shape tiles, *Pipe* may not be beneficial if the required data cannot be staged in on-chip buffer. *Para* is similar to *Pipe* but is only applicable to tiles without data dependency. By contrast, *Seq* provides no improvement for latency, but it saves on-chip compute and memory resources because only one tile can be executed at a time. *Shar* is similar to *Seq* but allows more data staged in on-chip buffer, thus increasing the data locality at the cost of memory usage.

Third, we explain the loop tiling briefly. Loop tiling is the most used optimization technique in mapping design and performance tuning [20, 26, 49]. A loop tiling choice is composed of two parts: the selection of loops to tile, and the selection of tiling factors. As shown in Figure 4 part e), the selection of loops to tile affects the computation granularity. Choosing more loops for tiling gives a fine-grained dataflow while choosing fewer loops for tiling makes a coarse-grained dataflow. The selection of tiling factors is to maximize the performance with respect to hardware resource constraints by balancing data load/store latency and computation latency.

4.2 Tile-centric Notation

To express the fusion dataflow design choices in the 3D space, we propose a tile-centric notation. In the notation, a tile (T_n) at memory level n is defined as

$$T_n = \{l_1^n, l_2^n, ...\}(T_{n-1}^1, T_{n-1}^2, ...) \quad (1)$$

where $\{l_1^n, l_2^n, ...\}$ is a loop nest over a list of sub-tiles ($T_{n-1}^1, T_{n-1}^2, ...$), forming a tree structure with this recursive definition. The notation naturally aligns with the fusion dataflow structure. For compute ordering, different ordering trees are expressed using the tile definition above. For resource binding, primitives can be added to both loops (intra-tile) and tiles (inter-tile). For loop tiling, the loops in each tile correspond to the tiling results and thus express the tiling choices naturally.

We show how to use our tile-centric notation to express a fusion dataflow for the example in Figure 4 part a). One possible fusion dataflow is as follows

Tile Definition (Ordering and Tiling):

$$level\ 0: T_0^0 = \{i_0, l_0, k\}(A),\quad T_0^1 = \{i_0, l_0\}(B),\quad T_0^2 = \{i_0, j_0, l_0\}(C)$$
$$level\ 1: T_1^0 = \{i_1, l_1\}(T_0^0, T_0^1),\quad T_1^1 = \{i_1, j_1, l_1\}(T_0^2)$$
$$level\ 2: T_2^0 = \{i_2, j_2, l_2\}(T_1^0, T_1^1)$$

Inter-tile (Binding): **Pipe**(T_0^0, T_0^1), **Shar**(T_1^0, T_1^1)

Intra-tile (Binding): **Sp**(i_2), **Sp**(i_1), **Sp**(i_0)

This dataflow uses the same ordering in Figure 4 part c). For inter-tile, T_0^0, T_0^1 form a pipeline; T_1^0, T_1^1 use *Shar* primitive. If the inter-tile primitive is not specified, TileFlow uses *Seq* by default. For intra-tile, loops i_0, i_1, i_2 are mapped to spatial, while other loops are mapped to temporal by default.

5 TREE-BASED ANALYSIS

From our tile-centric notation, we can get a tree representation of a fusion dataflow. We call such a tree structure an analysis tree. In this Section, we introduce the tree-based analysis of TileFlow for data movement volume and resource usage. We also explain how to calculate performance results using these metrics.

5.1 Data Movement Analysis

Data movement analysis is used to calculate the amount of data moved between different levels of on-chip memory. Different fusion dataflows with different compute orders, resource bindings, and tiling decisions may result in different data reuse and thus different data movement volume. To calculate the data movement volume, we start from a simple case: only one tile with perfectly nested loops in the analysis tree. Then, we consider a more complex case for the data movement between two tiles.

5.1.1 Single Tile Data Movement Analysis. A single tile node in an analysis tree (without any children) represents a perfect loop nest (polyhedron). There are two types of loops in the tile: spatial loops and temporal loops. For each iteration step t in the tile, for each tensor Z, a slice of data is read or updated. We denote this as

$$Slice_Z^t = Z[b_0^t : e_0^t,\ b_1^t : e_1^t, ...,\ b_{D-1}^t : e_{D-1}^t]$$

where b_i^t is the slice beginning address of dimension i and e_i^t is the slice ending address (exclusive) of dimension i. For each dimension i, although the beginning address b_i^t and ending address e_i^t vary with different time steps t, the extent of the slice $(e_i^t - b_i^t)$ remains constant and is determined by the spatial loops at dimension i. For example, we use a batched 1D convolution in Figure 5 to explain the data slice. For tensor A in this example, in different execution time steps, although different sub-matrices of A are used in computation, all the sub-matrices have the same size of 4×6.

Single tile data movement happens between adjacent time steps (as the proceeding of temporal loops), when a new slice of data is required for the next step computation and the old slice of data is no longer needed. Formally, for each time step t, for tensor Z, the data movement volume is

$$DM_Z^t = |Slice_Z^t - Slice_Z^{t-1}|$$

Note that $Slice_Z^t$ is a set of data, so $Slice_Z^t - Slice_Z^{t-1}$ is the set difference operation, the result of which is the set of data required by time step t but not available in time step $t-1$. $|.|$ is norm operator, calculating the number of elements in a set, so DM_Z^t is a non-negative integer value. The total data movement volume DM_Z is

$$DM_Z = \sum_{t_i \in Bounds} U_i$$

$$\text{where } U_i = \begin{cases} |l_i| \times (DM_Z^{t_i} + (|l_{i-1}| - 1) \times U_{i-1}), & i > 1 \\ DM_Z^{t_0}, & i = 0 \end{cases}$$

where $Bounds$ is the set of time step boundaries t_i ($0 \le i \le Dim$, Dim is the number of temporal loops) for temporal loops. A time step boundary is the time step when a temporal loop is at its upper bound and is going to return to its lower bound. In Figure 5, we show the data movement volume of each tensor from time step $[0, 0]$ to time step $[0, 1]$. We use a vector to express time step for simplicity so that we can correlate each value in the time step vector to the temporal loops (in this example, we have two temporal loops i_1, j_1, so the time vector has two values). Tensor A needs 4×4 new elements for the next step computation and reuses 4×2 elements from previous step, tensor B needs 4×3 elements, while tensor C is fully reused and no data movement is needed. We also show the time step boundaries in Figure 5 (t_0 =[0, 0], t_1 =[0, 2], and t_2 =[2, 0]). Note that time step [0, 0] is also a boundary for compulsory data miss. The total data movement volume for tensor A in this example is 168 (elements).

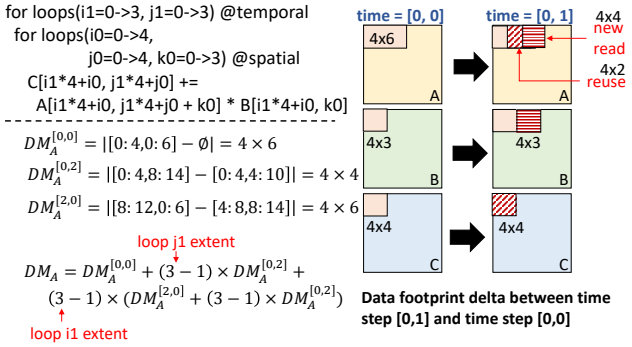


Figure 5: Single tile data movement analysis.

5.1.2 Inter-tile Data Movement Analysis. For an analysis tree with a complex hierarchy, a tile will have children tiles. To calculate the data movement volume for the parent tile, we use a similar approach to that of a single tile by calculating the data slice set difference between two adjacent time steps for all the children tiles. In a given step t, for each child tile, we first calculate its data slices for all the input/output tensors, then we arrange these child tiles in a list according to the time steps (decided by the temporal loops of parent tile). For every two adjacent tiles, the set difference is calculated to obtain how much data movement is required when switching from one tile to another tile. By accumulating all the set differences together, we will finally get the data movement of the parent tile. We visualize this approach in Figure 6 (top part) using a simple example, where the parent tile (Tile 0) only has two temporal loops. We list the execution sequence of children tiles (Tile 1 and Tile 2) according to the time steps. Set difference of data slices is done for every two adjacent execution steps and the final data movement volume is the summation of all the set difference results. In implementation, there is no need to fully unroll all the time steps. Thanks to the regularity of DNN workloads, we only need to consider time step boundaries, which is similar to single tile analysis.

Different inter-tile resource binding decisions may affect the data movement volume. For *Shar*, *Para*, and *Pipe*, the data movement calculation approach is the same as above. But for *Seq*, after the execution of a tile, its accessed data slices will be evicted unless they are needed by the following tile. This will increase data movement during execution. We model this in TileFlow by clearing the data slice of one tile for its tensors that are not used by the following tile.

In previous analysis, the data movement happens within one level of memory. Two tiles in the analysis tree may also reside in different levels of memory. For such a case, we need to calculate the data movement volume between memory levels. In Figure 6, we show two tiles (Tile 1 in level X and Tile 2 in level Y) in the bottom part. We can first analyze their data movement volume separately to obtain how much data Tile 2 needs from Tile 1. According to the

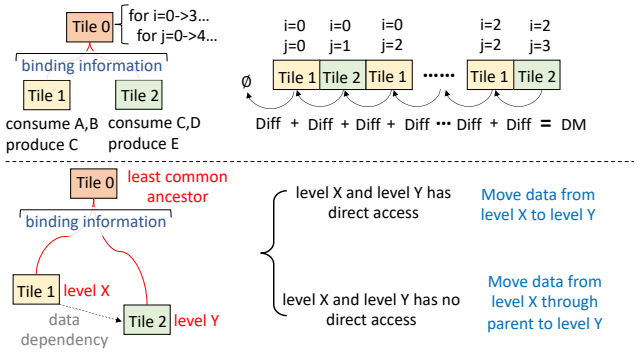


Figure 6: Inter-tile data movement analysis.

architecture design, there are two different cases. If level X memory can't move data to level Y memory directly (which is common in DNN accelerators [36]), then we need to move the data through their least common ancestor tile (Tile 0). Otherwise, we can directly move data between Tile 1 and Tile 2 and record the data movement volume between the memory of level X and level Y. This difference is modeled in TileFlow and the corresponding data movement volume is recorded to the memory levels where the movement happens.

5.2 Resource Usage

By traversing the analysis tree, we can also calculate resource usage (memory and compute). For each tile T_n in the analysis tree, if it has no more than one sub-tile (so it's a perfect loop nest), the number of PEs and memory footprint it uses can be calculated by polyhedron analysis [31, 38, 45]. If there are more than two sub-tiles, for each pair of sub-tiles T^1_{n-1}, T^2_{n-1}, the number of PE used by the parent tile depends on the inter-tile resource binding primitive. The calculation formula is

$$NumPE(T_n) = \begin{cases} max(NumPE(T^1_{n-1}), NumPE(T^2_{n-1})), & \text{for } Seq \text{ and } Shar \\ NumPE(T^1_{n-1}) + NumPE(T^2_{n-1}), & \text{otherwise} \end{cases}$$

The memory footprint also depends on resource binding decisions. The formula is

$$FootPrint(T_n) = \begin{cases} max(Footprint(T^1_{n-1}), Footprint(T^2_{n-1})), & \text{for } Seq \\ Footprint(T^1_{n-1}) + Footprint(T^2_{n-1}), & \text{otherwise} \end{cases}$$

5.3 Performance Estimation

To calculate performance metrics (latency and energy) in TileFlow, we need hardware architecture specifications *Spec*. For each tile T_n at level n, it has three execution phases: data loading, computation, and data storing. Data loading and storing volume is obtained through data movement volume analysis introduced in previous Sections and the data loading/storing latency is estimated by dividing the data movement volume by the bandwidth BW_n (from *Spec*) of the corresponding level of memory. The computation latency is computed as follows

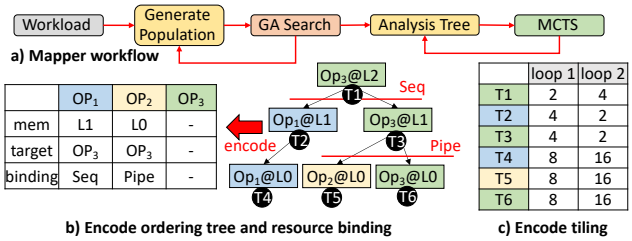


Figure 7: Workflow and encoding of TileFlow mapper.

$$\text{Lat}_{T_n} = \begin{cases} Perfect\_Tile\_Latency(T_n, Spec), & \text{if } T_n \text{ has no sub-tiles} \\ Max\{\frac{DM^{load}_n}{BW_n}, \sum_i \{Lat_{T^i_{n-1}}\}, \frac{DM^{store}_n}{BW_n}\}, & Seq \text{ or } Shar \\ Max\{\frac{DM^{load}_n}{BW_n}, \max_i \{Lat_{T^i_{n-1}}\}, \frac{DM^{store}_n}{BW_n}\}, & \text{otherwise} \end{cases}$$

where $Perfect\_Tile\_Latency$ is to calculate the latency for a perfectly nested tile with polyhedron-based approach [31, 38, 45]. We assume the data load, execution, and data store are fully pipelined with double buffer. For energy estimation, we use existing energy estimation frameworks [45, 64] by passing them the total number of memory access operations (which are results of our data movement analysis introduced in previous Sections) and computation operations (which is an inherent property of the workloads and can be calculated from the input workloads).

6 TILEFLOW MAPPER DESIGN

Manually finding efficient fusion dataflows is challenging considering the intractable design space. We design a simple mapper in TileFlow to help design dataflows. The mapper uses a combination of genetic algorithm and Monte Carlo Tree Search (MCTS) [56] in exploration. The workflow is shown in Figure 7 part a). Other more complicated exploration approaches [6, 20, 24, 25, 76, 80] are also applicable to TileFlow. We leave this for future work.

The combined algorithm works as follows. We first encode different ordering trees and binding primitives into Tables as shown in Figure 7 part b). Each operator corresponds to one column; the entry *mem* represents which level of memory to stage intermediate data; *target* represents which operator to fuse into; *binding* represents the binding primitive. In the example, OP_1 is fused to OP_3 at memory *L1* and OP_2 is fused to OP_3 at memory *L0*, forming an analysis tree. The genetic algorithm can generate a population of analysis trees from a set of randomly sampled initial choices through crossover and mutation of the encoded choices.

Second, all the generated analysis trees are passed to the MCTS algorithm to find optimized tiling factors within hardware resource constraints. For each step, it selects one loop and assigns it a tiling factor within its trip counts. Then, it updates the constraints using the known factor and pass the new constraints to the next untiled loop. In Figure 7 part c), we show an example of encoding tiling, which is a Table for two loops. Each searching node in the Monte Carlo Tree corresponds such a tiling table, which records the chosen tiling factors for each loop in each tile. When the MCTS reaches a leaf node (that is, all the factors in the tiling table are

decided), a complete fusion mapping is produced and evaluated using TileFlow model. The results are feedbacks to MCTS to update upper confidence bounds (UCB) for later searching.

By repeating such search steps hundreds of times, the optimized tiling factors are returned to the genetic algorithm. Finally, the genetic algorithm uses the best tiling factors to obtain the best performance for each analysis tree. The top-K (K is a parameter) analysis trees are reserved to produce the next population. The above steps are repeated hundreds of times, and the best analysis tree is returned as the best fusion dataflow.

7 EVALUATION

7.1 Model Validation

We first validate the model accuracy of TileFlow. We use both real hardware design and state-of-the-art performance model for validation. For performance model, we use Timeloop [45] for comparison. For real hardware, we implement a TPU-derived accelerator to compare the predicted cycle of TileFlow with the RTL-level simulation result of the accelerator.

Accelerator Implementation: We implement the accelerator in Chisel to generate Verilog RTL. The accelerator has four cores. Each core has two PE arrays: one for matrix multiplication (16×16) and the other for vector computation (16×3). The on-chip buffer size is 384KB per core. The DRAM bandwidth is 25.6GB/s. The word width is 16 bits. The RTL is then synthesized using Cadence Genus Synthesis and Innovus tool. The synthesis reports show that our accelerator area is $7.84m^2$ under TSMC 22nm technology, and the frequency is 400 MHz. The accelerator supports matrix, vector, load, and store instructions. We program test cases using the instructions and compile them into binary. At last, we use Verilator [58] (version 4.0) to simulate the RTL and binary to get runtime performance (cycle) in evaluation.

For comparison with Timeloop, we use a single operator workload because Timeloop doesn't support multi-operators or fusion. We use matrix multiplication and enumerate 1152 different mappings for it. We compare the predicted cycle results of TileFlow and Timeloop as shown in Figure 8a. The x-axis is the reported cycle of Timeloop, and the y-axis is the reported cycle of TileFlow. The correlation between the results of TileFlow and Timeloop is high ($R^2 = 0.999$). Similarly, for energy prediction, TileFlow's accuracy is still very high compared to Timeloop as shown in Figure 8b. The average absolute value error is 0.1%. For comparison with real accelerator, we use self-attention [10]. We program highly optimized fusion kernels for our accelerator in assembly and enumerate 131 different mappings (by changing tiling factors and shapes). We compare the runtime cycle of our accelerator and the results predicted by TileFlow as shown in Figure 8c. The x-axis represents different mapping cases, and the y-axis is the relative runtime cycle (TileFlow Cycle/Real Cycle). The yellow circles are the results of graph-based method [72], the blue triangles are results of TileFlow. The average error of TileFlow in absolute value is 5.4%, while the average error of graph-based method is 48.8%. Figure 8d is the comparison of TileFlow's energy results and real energy consumption. The average error in absolute value is 6.1%. For part of the test cases, TileFlow's energy prediction is not accurate. This is mainly because these cases use small tiling factors. TileFlow tends to overestimate data movement volume (i.e., more energy for data access) for them because it assumes data replacement happens for every outer iteration. But in real accelerator, small tiles may not cause data replacement. All these experiments demonstrate the high accuracy of TileFlow.

Table 2: The shape of self-attention.

Name	Model	*num_heads*	*seq_len*	*hidden*
Bert-S	Bert	8	512	512
Bert-B	Bert	12	512	768
Bert-L	Bert	16	512	1024
ViT/14-B	ViT	12	256	768
ViT/14-L	ViT	16	256	1024
ViT/14-H	ViT	16	256	1280
ViT/16-B	ViT	12	196	768
ViT/16-L	ViT	16	196	1024
ViT/16-H	ViT	16	196	1280
T5	T5	16	1024	1024
XLM	XLM	12	1024	768

Table 3: The shape of convolution chains.

Name	In_C	$Height$	$Width$	Out_C_1	Out_C_2
CC1	64	112	112	192	128
CC2	32	147	147	64	80
CC3	64	56	56	128	64
CC4	128	28	28	256	128
CC5	16	227	227	64	16

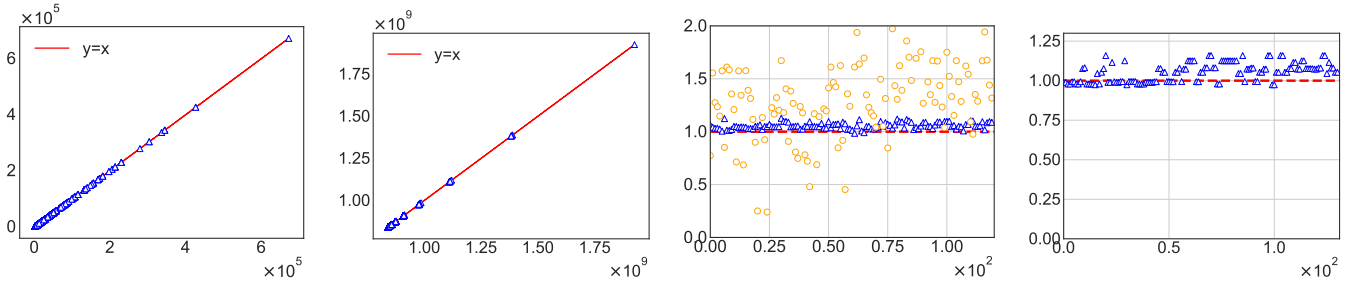
Table 4: Accelerator Specification in Evaluation.

Name	# of PEs	Sub-Core	Core	On-chip Mem. Size	DRAM BW.
Edge	32×32	1	4	L1: 4MB	60 GB/s
Cloud	256×256	16	4	L1: 20MB L2: 40MB	384 GB/s

7.2 Performance Exploration Results

In this part, we show the performance of the TileFlow mapper. We use the input shapes in Table 2. We implement five different fusion dataflows for self-attention in TileFlow using our tile-centric notations: *Layerwise*, *Uni-pipe*, *FLAT-HGran*, *FLAT-RGran*, and *Chimera* as shown in Table 5. For accelerator configuration, we use the Edge specification in Table 4. The Edge accelerator has four cores, each has 4MB L1 buffer. The L1 bandwidth is 1.2TB/s. To support the non-linear softmax layer in self-attention, we need to expand it into five small operators (*max*, *sub*, *exp*, *sum*, *div*). Each small operator is then converted to nested loops so that they can be handled by TileFlow.

Our experiment machine is Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz. We use a single thread in execution. We set the mapper to explore 50 rounds; each round samples 200 tiling choices and needs about 12 seconds for evaluation. In Figure 9 part a), we plot the normalized performance change as exploration proceeds. We use the Bert-S input shape in Table 2. The results show that the TileFlow



(a) Absolute cycle validation against Timeloop model. **(b) Absolute energy validation against Timeloop model.** **(c) Relative cycle validation with real accelerator.** **(d) Relative energy validation with real accelerator.**

Figure 8: Validation Results of TileFlow.

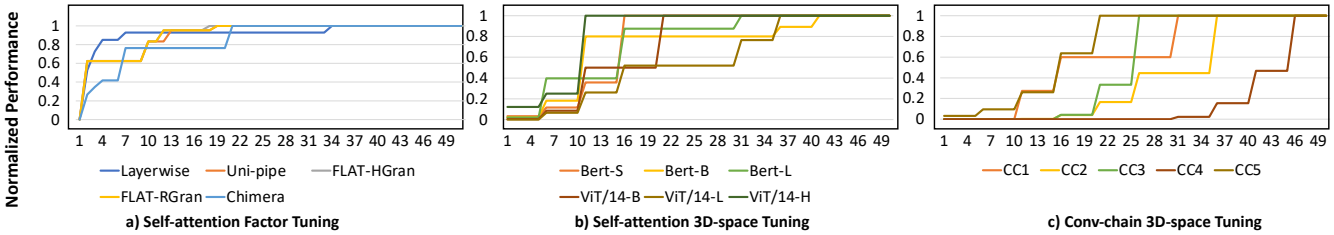


Figure 9: Performance of TileFlow Mapper

Table 5: Different dataflows in evaluation.

Name	Explanation
Self-attention Dataflows	
Layerwise	No fusion. Map one Op to hardware at a time.
Uni-pipe	Pipeline $Q \times K$ and softmax without tiling multi_heads/row.
FLAT-HGran [26]	Fuse $Q \times K$ and softmax and tile batch and multi_heads dimension.
FLAT-RGran [26]	Fuse $Q \times K$ and softmax and tile batch, multi_heads, and row dimension.
Chimera [79]	Fuse $Q \times K$ and softmax and tile all the dimensions.
Convolution Chain Dataflow	
Layerwise	No fusion. Map one convolution to hardware at at time.
Fused-Layer [2]	Fuse two convolutions with height and width dimensions tiled.
ISOS [70]	Fuse two convolutions with only width tiled.

mapper needs around 3.2 minutes to 6.4 minutes to converge. We then show the exploration results in the full 3D space (compute ordering, resource binding, and loop tiling). Besides self-attention layers, we also use convolution chains in evaluation. The input shapes of the convolution chains are shown in Table 3. The 3D space size is much larger than tiling space. For the workloads we use, the number of different datafows ranges from 5103 to 20412, each dataflow design further has its own tiling factor space. We plot the exploration traces in Figure 9 part b) and part c). The x-axis represents search rounds. TileFlow requires a few rounds (less than 50 rounds, each round samples 20 fusion dataflows) to converge.

To fully explore the 3D space requires a long time (1-2 days), we use multiprocessing to accelerate the exploration (56 processes) and the exploration time is reduced within one hour. The exploration results show that different input shapes prefer different dataflows. It's hard to choose one dataflow that achieves the best performance for all the input shapes. As a result, we choose one dataflow that gives the geometric optimal performance over all input shapes as a representative of TileFlow (referred to as *TileFlow* dataflow). For self-attention, the *TileFlow* dataflow is to pipeline all the three computation stages: $Q \times K$, softmax, and $L \times V$ (symbols used in Figure 1) with all the loops tiled. The speedup of this dataflow to *Layerwise* self-attention dataflow is 6.65×. Compared to the best self-attention dataflow (*FLAT-RGran*) in Table 5, the average speedup is 1.85×. For convolution chains, the *TileFlow* dataflow is to pipeline the two convolutions with their channel dimensions tiled. The speedup of this dataflow to *Layerwise* is 1.31×. The speedup to *Fused-Layer* is 1.28×.

7.3 Fusion Dataflow Comparison

In this part, we compare the state-of-the-art fusion dataflows [2, 26, 70, 79] with the dataflows of TileFlow (described in Section 7.2) for latency and memory access. We use two accelerator specifications: Edge and Cloud, as listed in Table 4. The Cloud specification has four cores. Each core further has 16 sub-cores and one 40MB L2 buffer. The L1 bandwidth is 9.6TB/s, the L2 bandwidth is 1.9TB/s. To ensure a fair comparison among different dataflows, we utilize TileFlow's mapper to determine the tiling factors for all the different dataflows.

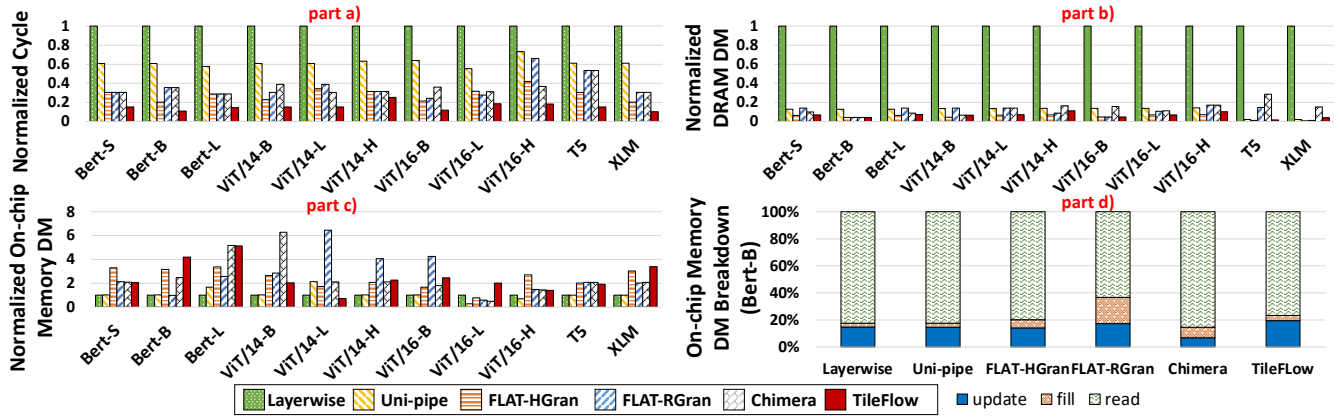


Figure 10: Fusion dataflow evaluation for self-attention on Edge accelerator (DM is data movement volume).

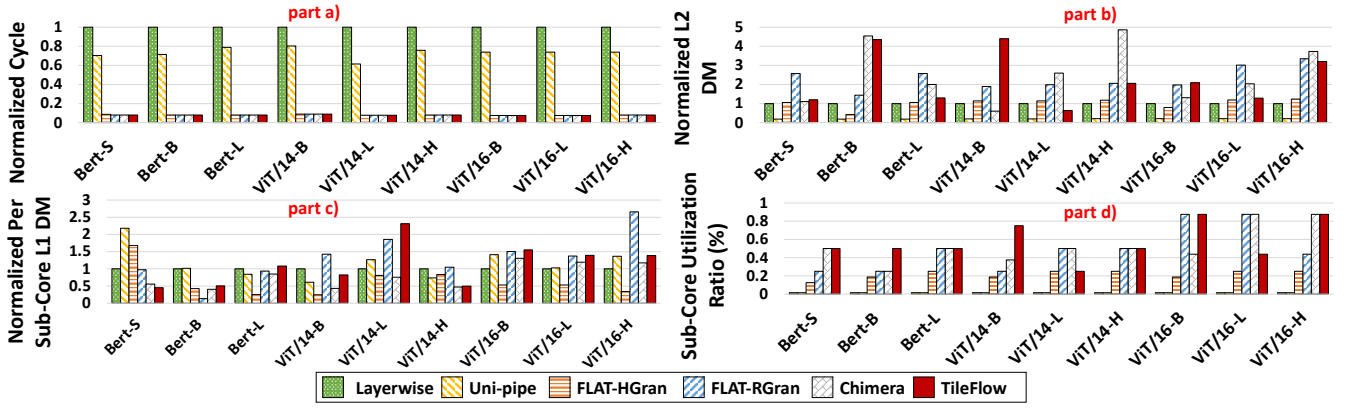


Figure 11: Fusion dataflow evaluation for self-attention on Cloud accelerator (DM is data movement volume).

First, we show the evaluation results on Edge accelerator in Figure 10. In part a), we show the normalized cycle of all the dataflows for all the self-attention input shapes. Overall, compared to *Layerwise*, *Uni-pipe* dataflow achieves 1.62× speedup, *FLAT-HGran* dataflow achieves 3.59× speedup, *FLAT-RGran* dataflow achieves 2.89× speedup, and *Chimera* dataflow achieves 2.91× speedup. TileFlow achieves the best performance compared to all the other dataflows: 6.65× speedup to *Layerwise*; 4.11× speedup to *Uni-Pipe*; 1.85× speedup to *FLAT-HGran*; 2.30× speedup to *FLAT-RGran*; 2.28× speedup to *Chimera*. To analyze the source of speedup, we also show the data movement volume (DM) results of DRAM and on-chip memory in part b) and part c) in Figure 10. On average, compared to *Layerwise* dataflow, *Uni-pipe* and *FLAT-HGran* dataflows can reduce 90.4% DRAM access, *FLAT-RGran* dataflow can reduce 81.5% DRAM access, *Chimera* dataflow can reduce 75.1% DRAM access, *TileFlow* dataflow can reduce 87.1% DRAM access. The reduction of DRAM access implies a higher data reuse in on-chip memory. Although *FLAT-HGran* has less DRAM data movement than *TileFlow* for some of the input configurations, its PE utilization is not high (about 50% of *TileFlow*). So its performance is lower. We plot the on-chip memory (L1 for Edge) data movement volume in part c). For the best cases of all the dataflows, L1 data movement volume is increased by 2.01 × −6.45× for the 11 input shapes. For a specific input shape (Bert-B), we show the detailed L1 data movement breakdown in Figure 10 part d). The *update* refers to the write back to L1 buffer, *fill* refers to the initial data load from DRAM to L1 buffer, *read* refers to the data load from L1 buffer to register. On average, 80.9% of the L1 data movement is *read*, 14.7% is *update*.

Comparing all the dataflows, for Edge accelerator, *Uni-pipe* is better than *Layerwise* dataflow because the fusion eliminates a large number of DRAM access; *FLAT-HGran* dataflow is better than *Uni-pipe* dataflow because of tiling, the tiled blocks are spatially mapped to different cores and increase parallelism; *FLAT-RGran* and *Chimera* dataflows produce similar performance to *FLAT-HGran*, but their L1 buffer footprint is smaller, *FLAT-RGran* only requires 28.4% of the L1 buffer size that *FLAT-HGran* uses for computation, while *Chimera* only requires 14.8%.

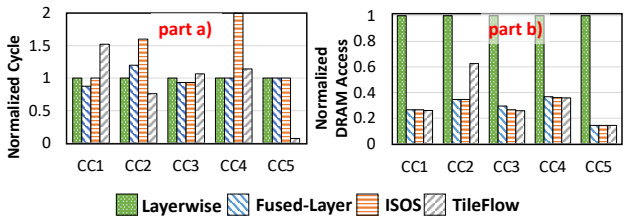


Figure 12: Fusion dataflow evaluation for 3×3 convolution chain on Cloud Accelerator.

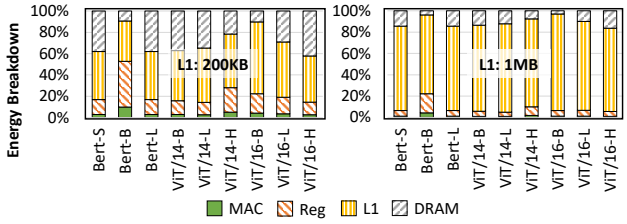


Figure 13: Energy breakdown for *FLAT-RGran* dataflows on Edge accelerator for Bert-S.

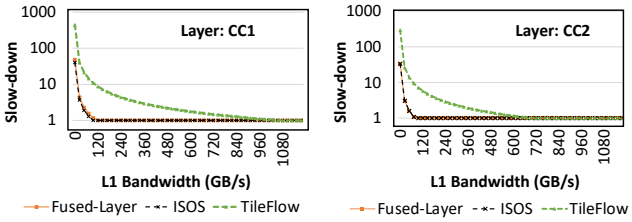


Figure 14: Slow-down of L1 on Edge accelerator under different bandwidth configurations for convolution chains.

Then, we show the evaluation results on Cloud accelerator in Figure 11. In part a), we show the normalized runtime cycle results. Compared to *Layerwise* dataflow, the speedup of *Uni-pipe* dataflow is 1.37×, the speedups of all the other dataflows are the same: 12.63×. The speedup of *Uni-pipe* is low due to low spatial utilization. *Uni-pipe* only uses around 25% of the spatial cores for computation for lack of tiling, while other dataflows can fully utilize all the spatial cores with tiling. *FLAT-HGran*, *FLAT-RGran*, *Chimera* can achieve the same best performance. This implies that for Cloud accelerator (both compute and bandwidth resources are abundant), the tiling granularity has little affect on performance. TileFlow's mapper is always able to find the optimized tiling factors to maximize the performance for different tiling granularity.

We also show the on-chip memory data movement volume (DM) in Figure 11 part b) and part c). All the fusion dataflows have the same amount of DRAM access reduction ratio (geometric mean is 86.6% reduction) compared to *Layerwise* dataflow. So we focus on on-chip memory data movement. In part b), we show the L2 buffer data movement volume. All the dataflows except *Uni-pipe* have larger amount of data movement volume, showing a higher data reuse ratio in on-chip memory. *Uni-pipe* has low L2 data movement because its data is largely staged in L1 buffer. For L1 buffer data movement volume, we show the statistics for one sub-core in Figure 11 part c). For fusion dataflows, although there is no L1 data movement increase for single sub-core compared to *Layerwise* dataflow (about 52.1% – 108.5% that of *Layerwise*), the total amount of L1 data movement for all the sub-cores has increased significantly (by 7.01 × –33.27×). We show the sub-core spatial utilization ratio in Figure 11 part d). Compared to *Layerwise*, the utilization ratio of *FLAT-HGran* is improved by 13.46×; *FLAT-RGran* improves the utilization ratio by 28.34×; *Chimera* improves the ratio by 32.02×; *TileFlow* improves it by 34.58×.

Besides self-attention, we also show the dataflow evaluation results for convolution chains using the input shapes in Table 3 on Cloud accelerator. These input shapes are from real networks [19, 50, 59] and follow the setting used in Chimera [79]. The two convolutions in the convolution chain uses 3×3 filter size. The normalized runtime cycle and DRAM access are shown in Figure 12. The geometric mean speedup of *Fused-Layer* dataflow to *Layerwise* is 1.01×. Although *Fused-Layer* dataflow brings little latency improvement, it reduces DRAM access by 73.0% and reduces energy consumption by 30.1%. *ISOS* fails to provide speedup. *ISOS* is originally designed for sparse CNN. But we use it for dense CNN in experiments. *TileFlow* achieves 1.59× speedup to *Layerwise* and *Fused-Layer*.

7.4 Energy Breakdown

We show the energy and bandwidth evaluation results of TileFlow. First, we evaluate all the fusion dataflows for self-attention on the Edge accelerator. On average, compared to *Layerwise*, *Uni-pipe* can save 15.4% energy, *FLAT-HGran* can save 16.3% energy, *FLAT-RGran* can save 8.7% energy, *Chimera* can save 9.1% energy, and *TileFlow* dataflow (described in Section 7.2) can save 13.3% energy. To analyze the energy breakdown, we evaluate *FLAT-RGran* for Bert-S on Edge accelerator with different L1 buffer sizes as shown in Figure 13. L1 energy consumption accounts for most of the total energy consumption. The SRAM buffer size dictates the read/write energy of L1 buffer. We use two different buffer sizes for L1: 200KB and 1MB. For small L1 buffer, on average, 46.5% energy is used for L1 access, 33.3% energy is used for DRAM access, 16.5% energy is used for register access. For large L1 buffer, on average, 80.1% energy is used for L1 access, 12.3% energy is used for DRAM access, 6.1% energy is used for register access.

7.5 Sensitivity Study

Bandwidth: We first study TileFlow's sensitivity to bandwidth. To do this, we choose Edge accelerator and enumerate L1 bandwidth from 1GB/s to 1200GB/s by step 1GB/s. The metric we use for bandwidth is slow-down [45]:

$$\text{Slow-down} = max\{\frac{\text{L1 access latency}}{\text{L1 compute latency}}, 1\}$$

If the slow-down of L1 is larger than 1, it indicates that L1 access dominates the runtime performance and the workload become memory-bound. So the suitable L1 bandwidth is the minimal value that makes L1 slow-down as 1. In Figure 14, we show the trace for two layers: CC1 and CC2. The results show that *Fused-Layer* and *ISOS* are similar in bandwidth requirements. The suitable bandwidth for them is 96GB/s. *TileFlow* dataflow (described in Section 7.2) is

Table 6: Performance (10^6 cycles) of TileFlow for different PE array sizes.

PE Size	8^2	16^2	32^2	64^2	128^2	256^2
baseline	12.58	3.15	2.36	1.73	1.57	1.57
TileFlow	6.29	1.57	1.57	1.57	1.57	1.57

Table 7: Evaluation of different FLAT dataflows for T5 (batch 128) on Cloud accelerator with/without tiling exploration.

Part a) No Tiling Exploration. No Memory Limit.					
Dataflow	**MGran**	**BGran**	**HGran**	**RGran**	**TileFlow**
Cycle (10^6)	335.54	151.00	67.11	18.87	16.78
L1 Used (MB)	131.14	4.10	0.26	0.17	0.01
L2 Used (MB)	262.14	16.38	4.10	2.18	0.19
Part b) With Exploration for Tiling. No Memory Limit.					
Dataflow	**MGran**	**BGran**	**HGran**	**RGran**	**TileFlow**
Cycle (10^6)	335.54	8.39	8.39	8.39	7.67
L1 Used (MB)	131.14	65.57	8.20	2.11	0.16
L2 Used (MB)	262.14	131.07	131.07	196.61	131.07
Part c) With Exploration for Tiling. With Memory Limit.					
Dataflow	**MGran**	**BGran**	**HGran**	**RGran**	**TileFlow**
Cycle (10^6)	OOM	OOM	14.68	14.68	16.78
L1 Used (MB)	-	-	4.10	0.53	0.05
L2 Used (MB)	-	-	32.77	12.29	20.48

more sensitive to both L1 bandwidth and layer input shape. The suitable L1 bandwidth for CC1 is 1080 GB/s, and the suitable bandwidth for CC2 is 720 GB/s.

PE Size: Second, we evaluate TileFlow's dataflow using different PE array sizes from 8×8 to 256×256 with step 2 for each dimension. The workload is self-attention (the shape is Bert-Base). We show the results in Table 6. The baseline is *FLAT-RGran*. The results show that TileFlow can adapt to different PE array sizes and converge to a stable optimal performance when PE size is larger than 16×16. The speedup to baseline is also stable for small PE sizes (around 2×).

Tiling: Third, we evaluate the sensitivity of TileFlow to tiling granularity and tile factors. FLAT proposes four different tiling granularities: MGran (no tiling), BGran (tiling batch), HGran (tiling batch and multi_heads), and RGran (tiling batch, multi_heads, and rows). In this part, we set batch size to 128 and compare the performance of the four granularities of FLAT. We also analyze the effect of tiling factor exploration in this part. The workload is self-attention with T5 configuration. We use Cloud accelerator for evaluation.

The results are shown in Table 7. When we use fixed tiling factors for the dataflows without exploration for tiling (Table 7 part a), we find that the finer the tiling granularity, the better the performance and the less the on-chip memory required for execution. For a fair comparison, the tiling factors for batch dimension are the same for *FLAT-BGran*, *FLAT-HGran*, *FLAT-RGran*, and *TileFlow*; the tiling factors for multi_heads are the same for *FLAT-HGran*, *FLAT-RGran*, and *TileFlow*; the tiling factors for row dimension are the same for *FLAT-RGran* and *TileFlow*. The fixed factors are not efficient because of the low on-chip memory utilization and spatial utilization.

When exploring tiling factors for all the dataflows, we evaluate two different scenarios. The first is to ignore the on-chip buffer capacity limit (unlimited on-chip memory resources). TileFlow's mapper first find the optimal tiling factors for all the dataflows. Then, Using the optimal factors, we infer the amount of on-chip memory resources required by each dataflow. The results (Table 7 part b) show that without memory limit, *FLAT-BGran*, *FLAT-HGran*, and *FLAT-RGran* can achieve the same performance. These dataflows require less L1 memory than the Cloud accelerator has provided (20 MB), while requiring from 3.3× to 4.9× more L2 memory than provided (40 MB). The second scenario is to take memory capacity constraints into consideration. The results (Table 7 part c) show that *FLAT-MGran* and *FLAT-BGran* both exceed memory limit. *FLAT-HGran* and *FLAT-RGran* still achieve the same performance. But they require different amount of on-chip memory. For example, the L2 memory consumption of *FLAT-RGran* is only 37.5% that of *FLAT-HGran*. For both scenarios, tiling exploration consistently achieves better performance than fixed tiling factors.

Compared to FLAT, TileFlow achieves similar or better performance at a lower cost of on-chip memory. The reason is that TileFlow tiles all the dimensions of all the three operators in self-attention ($S = Q \times K$, $L = Softmax(S)$, $A = V \times L$) thanks to the expressiveness of the tile-centric notation and the comprehensive exploration of full 3D design space (Section 4). Specially, for part c) in Table 7, tensors Q, K, L, V, and A all contribute to the L1 usage. TileFlow can tile both the row dimension and column dimension of S, L, and A. Tiling column dimensions will influence the L1 tile size of tensors K, L, and A. This tiling strategy is not explored by FLAT because FLAT does not tile column dimension of S, L, and A. FLAT requires at least one full row of intermediate data or output data to be staged in on-chip buffer. Each row of K and L (the length is 1024) has to be placed in L1 buffer, while TileFlow tiles the column dimension and decides the tiling factor by exploring the tiling space. In this case, the searched tiling factor for column dimension is 64, so only a sub-row of size 64 will be placed in L1 buffer. As a result, *TileFlow* can reduce the L1 usage, leading to an order of magnitude lower L1 usage.

In summary, finer tiling granularity is suitable for memory-limited scenarios. Different granularities can achieve similar performance when on-chip memory resource is abundant. Tiling exploration can always improve performance compared to fixed tiling factors.

7.6 Evaluation on GPU

TileFlow's dataflow can be integrated into machine learning frameworks such as PyTorch [46], TensorFlow [1], and TVM [5]. We use TVM as an exmpale. We use TVM's code generator to generate CUDA kernels on A100 GPU for *TileFlow* dataflow. We also implement *FLAT-RGran* dataflow using TVM as our baseline. For workloads, we use the self-attention layers from T5-Large [48] and XLM [8] with large input shapes (with seq_len from 1k to 256k). The results are listed in Table 8. *TileFlow* achieves better performance because of better tiling of intermediate softmax operator. *FLAT* doesn't tile the rows of softmax, so for 256k seq_len, its results are out of (shared) memory. *TileFlow* dataflow can support all the input shapes without any problem thanks to the proper tiling of softmax operator.

Table 8: Evaluate TileFlow' runtime (ms) on A100 GPU for self-attention layers of T5 and XLM with different seq_len.

Model	seq_len	1k	4k	16k	64k	256k
T5	baseline	1.13	16.58	156.99	1064.63	OOM
	TileFlow	0.23	3.10	47.75	756.99	12204.08
XLM	baseline	0.89	12.55	117.89	798.69	OOM
	TileFlow	0.23	2.45	35.96	567.69	9159.60

7.7 Discussion

TileFlow is mainly designed for dense workloads. Prior works such as GCNAX [33], OMEGA [14], Gamma [71], Spada [35], and Flexagon [43] support sparse workloads through customized PE arrays or caches. SparseLoop [66] proposes to use sparse acceleration features to model sparse architectures and workloads. This is also applicable to TileFlow, which is left for future work.

8 RELATED WORK

DNN Accelerators and Mappers: Various accelerators have been proposed [7, 12, 15, 18, 23, 29, 33, 36, 39, 41, 47] in recent years. These accelerators employ different dataflows for DNN acceleration. To exploit the high performance and energy-efficiency of these accelerators, different hardware mappers [20, 21, 24, 62, 74, 78, 81] have been proposed. For example, ConfuciuX [24] uses reinforcement learning to help search optimized hardware resource assignments and datflow styles. CoSA [21] proposes to use mixed integer programming to optimize mapping for spatial architecture. SARA [74] can compile general programs to reconfigurable dataflow accelerator with high parallelism and efficiency. The exploration methods in these works are orthogonal to TileFlow and can be implemented in TileFlow mapper.

Performance Models: Performance models [31, 38, 45, 54, 65, 66, 68] are important to both dataflow exploration and architecture design. Timeloop [45] uses spatial/temporal notations to describe mappings and can analyze latency, energy, and reuse for different customized architectures. MAESTRO [31] uses data-centric notations for dataflow description and calculates performance metrics through iteration analysis. Scale-SIM [54] provides systematic performance modeling for systolic array architecture. Interstellar [69] proposes to use Halide [49] schedule primitives to design DNN accelerators. TENET [38] proposes a relation-centric notation and uses polyhedral approaches for latency estimation. Sparseloop [66] provides a performance model for sparse workloads by modeling the sparsity and format for computation tiles. These models focus on single operator acceleration and mainly use polyhedron-based approach to infer data movement volume and reuse. For multi-operators, performance models are more necessary because of the complicated inter-operator data movement and deep accelerator memory hierarchy. MAGMA [25], NNest [28], Dnn-chip Predictor [75], HDA [32], HASCO [67], and H2H [72] analyze the whole DNN performance by separately evaluating each layer using single-operator performance models and then assemble the results to predict the performance of the whole DNN. They lack a detailed analysis of inter-operator on-chip data movement. Moreover, on-chip resource constraints are not considered for operator fusion optimizations.

Fusion Optimizations and Dataflows: As the bandwidth resource becomes critical in DNN performance, more and more fusion strategies are proposed. Software fusion techniques [5, 6, 9, 22, 34, 40, 44, 49, 76, 77, 79, 80, 82] use different compilers and schedulers to fuse multiple operators into one kernel and leverage the hardware on-chip memory to exploit locality. Among them, Halide [49], TVM [5], FlexTensor [80], Ansor [76], Chimera [79] use schedule-based fusion techniques. They use manually designed templates, machine-learning cost models, or analytical models to guide fusion strategy exploration. But they don't provide hardware performance models. Hardware fusion techniques [2, 13, 26, 42, 63, 70] design fusion dataflows to map the fused layers to hardware accelerators. Fused-layer [2] proposes to fuse CNN layers together with height and width dimensions tiled. ISOSceles [70] proposes to fuse sparse CNN layers with only width dimension tiled. FLAT [26] proposes four different dataflows to fuse self-attention layer with softmax rows stationary in on-chip buffer. TileFlow can express these fusion dataflows and provide a fair comparison among them in terms of latency and energy.

9 CONCLUSION

Fusion is an importation optimization in DNN dataflow design. Previous performance models focus on single operator acceleration without consideration for fusion. We propose a framework TileFlow that models dataflows for operator fusion using tile-centric notations and a tree-based approach to analyze data movement volume and accelerator resource usage. TileFlow can estimate latency and energy consumption for different dataflows and architecture specifications. In evaluation, TileFlow's dataflow achieves 1.85× runtime speedup compared to state-of-the-art work on average for self-attention and 1.28× speedup for convolution chains.

ACKNOWLEDGMENTS

We thank all the anonymous reviewers and our shepherd for their insightful suggestions. This work is supported in part by the National Natural Science Foundation of China (NSFC) under grant No.U21B2017.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter A. Milder. 2016. Fused-layer CNN accelerators. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 22:1–22:12. https://doi.org/10.1109/MICRO.2016.7783725

[3] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[4] Jingwei Cai, Yuchen Wei, Zuotong Wu, Sen Peng, and Kaisheng Ma. 2023. Inter-layer Scheduling Space Definition and Exploration for Tiled Accelerators. In *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*, Yan Solihin and Mark A. Heinrich (Eds.). ACM, 13:1–13:17. https://doi.org/10.1145/3579371.3589048

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. 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 2018, Carlsbad, CA, USA, October 8-10, 2018*. 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[6] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. 3393–3404. http://papers.nips.cc/paper/7599-learning-to-optimize-tensor-programs

[7] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE J. Solid State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[8] Alexis Conneau and Guillaume Lample. 2019. Cross-lingual Language Model Pretraining. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 7057–7067. https://proceedings.neurips.cc/paper/2019/hash/c04c19c2c2474dbf5f7ac4372c5b9af1-Abstract.html

[9] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *CoRR* abs/2205.14135 (2022). https://doi.org/10.48550/arXiv.2205.14135 arXiv:2205.14135

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[11] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=YicbFdNTTy

[12] 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 *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, Deborah T. Marr and David H. Albonesi (Eds.). ACM, 92–104. https://doi.org/10.1145/2749469.2750389

[13] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 807–820. https://doi.org/10.1145/3297858.3304014

[14] Raveesh Garg, Eric Qin, Francisco Muñoz-Martínez, Robert Guirado, Akshay Jain, Sergi Abadal, José L. Abellán, Manuel E. Acacio, Eduard Alarcón, Sivasankaran Rajamanickam, and Tushar Krishna. 2022. Understanding the Design-Space of Sparse/Dense Multiphase GNN dataflows on Spatial Accelerators. In *2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30 - June 3, 2022*. IEEE, 571–582. https://doi.org/10.1109/IPDPS53621.2022.00062

[15] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert J. Ou, Max Banister, Yakun Sophia Shao, Borivoje Nikolic, Ion Stoica, and Krste Asanovic. 2019. Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures. *CoRR* abs/1911.09925 (2019). arXiv:1911.09925 http://arxiv.org/abs/1911.09925

[16] Ross B. Girshick. 2015. Fast R-CNN. *CoRR* abs/1504.08083 (2015). arXiv:1504.08083 http://arxiv.org/abs/1504.08083

[17] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*. IEEE Computer Society, 580–587. https://doi.org/10.1109/CVPR.2014.81

[18] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro* 32, 5 (2012), 38–51. https://doi.org/10.1109/MM.2012.51

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[20] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W. Fletcher. 2021. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery Berger, and Christos Kozyrakis (Eds.). ACM, 943–958. https://doi.org/10.1145/3445814.3446762

[21] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. 2021. CoSA: Scheduling by Constrained Optimization for Spatial Accelerators. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 554–566. https://doi.org/10.1109/ISCA52012.2021.00050

[22] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 47–62. https://doi.org/10.1145/3341301.3359630

[23] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 1–12. https://doi.org/10.1145/3079856.3080246

[24] Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. 2020. ConfuciuX: Autonomous Hardware Resource Assignment for DNN Accelerators using Reinforcement Learning. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 622–636. https://doi.org/10.1109/MICRO50266.2020.00058

[25] Sheng-Chun Kao and Tushar Krishna. 2022. MAGMA: An Optimization Framework for Mapping Multiple DNNs on Multiple Accelerator Cores. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*. IEEE, 814–830. https://doi.org/10.1109/HPCA53966.2022.00065

[26] Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, Amir Yazdanbakhsh, and Tushar Krishna. 2023. FLAT: An Optimized Dataflow for Mitigating Attention Bottlenecks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 295–310. https://doi.org/10.1145/3575693.3575747

[27] Tero Karras, Samuli Laine, and Timo Aila. 2019. A Style-Based Generator Architecture for Generative Adversarial Networks. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 4401–4410. https://doi.org/10.1109/CVPR.2019.00453

[28] Liu Ke, Xin He, and Xuan Zhang. 2018. NNest: Early-Stage Design Space Exploration Tool for Neural Network Inference Accelerators. In *Proceedings of the International Symposium on Low Power Electronics and Design, ISLPED 2018, Seattle, WA, USA, July 23-25, 2018*. ACM, 4:1–4:6. https://doi.org/10.1145/3218603.3218647

[29] Kevin Kiningham, Philip Alexander Levis, and Christopher Ré. 2023. GRIP: A Graph Neural Network Accelerator Architecture. *IEEE Trans. Computers* 72, 4 (2023), 914–925. https://doi.org/10.1109/TC.2022.3197083

[30] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[31] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. *IEEE Micro* 40, 3 (2020), 20–29. https://doi.org/10.1109/MM.2020.2985963

[32] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. 2021. Heterogeneous Dataflow Accelerators for Multi-DNN Workloads. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 71–83. https://doi.org/10.1109/HPCA51647.2021.00016

[33] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan C. Bunescu. 2021. GCNAX: A Flexible and Energy-efficient Accelerator for Graph Convolutional Neural Networks. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 775–788. https://doi.org/10.1109/HPCA51647.2021.00070

[34] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. 2019. A coordinated tiling and batching framework for efficient GEMM on GPUs. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 229–241. https://doi.org/10.1145/3293883.3295734

[35] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 747–761. https://doi.org/10.1145/3575693.3575706

[36] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. 2021. Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : Industry Track Paper. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 789–801. https://doi.org/10.1109/HPCA51647.2021.00071

[37] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single Shot MultiBox Detector. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9905)*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer, 21–37. https://doi.org/10.1007/978-3-319-46448-0_2

[38] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. 2021. TENET: A Framework for Modeling Tensor Dataflow Based on Relation-centric Notation. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 720–733. https://doi.org/10.1109/ISCA52012.2021.00062

[39] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A Co-Design Framework for Enabling Sparse Attention using Reconfigurable Architecture. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. ACM, 977–991. https://doi.org/10.1145/3466752.3480125

[40] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 881–897.

[41] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision. *CoRR* abs/1803.04014 (2018). arXiv:1803.04014 http://arxiv.org/abs/1803.04014

[42] Jian Meng, Shreyas Kolala Venkataramanaiah, Chuteng Zhou, Patrick Hansen, Paul N. Whatmough, and Jae-sun Seo. 2021. FixyFPGA: Efficient FPGA Accelerator for Deep Neural Networks with High Element-Wise Sparsity and without External Memory Access. In *31st International Conference on Field-Programmable Logic and Applications, FPL 2021, Dresden, Germany, August 30 - Sept. 3, 2021*. IEEE, 9–16. https://doi.org/10.1109/FPL53798.2021.00010

[43] Francisco Muñoz-Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. Flexagon: A Multi-dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 252–265. https://doi.org/10.1145/3582016.3582069

[44] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 883–898. https://doi.org/10.1145/3453483.3454083

[45] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel S. Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019, Madison, WI, USA, March 24-26, 2019*. IEEE, 304–315. https://doi.org/10.1109/ISPASS.2019.00042

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*. 8024–8035. http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library

[47] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matthew Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Plasticine: A Reconfigurable Accelerator for Parallel Patterns. *IEEE Micro* 38, 3 (2018), 20–31. https://doi.org/10.1109/MM.2018.032271058

[48] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67. http://jmlr.org/papers/v21/20-074.html

[49] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 519–530. https://doi.org/10.1145/2491956.2462176

[50] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. 779–788. https://doi.org/10.1109/CVPR.2016.91

[51] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (Eds.). 91–99. http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks

[52] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-Resolution Image Synthesis with Latent Diffusion Models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*. IEEE, 10674–10685. https://doi.org/10.1109/CVPR52688.2022.01042

[53] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014*, Haizhou Li, Helen M. Meng, Bin Ma, Engsiong Chng, and Lei Xie (Eds.). ISCA, 338–342. http://www.isca-speech.org/archive/interspeech_2014/i14_0338.html

[54] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul N. Whatmough, Matthew Mattina, and Tushar Krishna. 2020. A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 23-25, 2020*. IEEE, 58–68. https://doi.org/10.1109/ISPASS48437.2020.00016

[55] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. 4510–4520. https://doi.org/10.1109/CVPR.2018.00474

[56] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nat.* 529, 7587 (2016), 484–489. https://doi.org/10.1038/nature16961

[57] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1409.1556

[58] Wilson Snyder. [n. d.]. Verilator. https://www.veripool.org/verilator/

[59] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2818–2826. https://doi.org/10.1109/CVPR.2016.308

[60] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 6105–6114. http://proceedings.mlr.press/v97/tan19a.html

[61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017), 5998–6008.

[62] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. 2021. UNIT: Unifying Tensorized Instruction Compilation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 77–89. https://doi.org/10.1109/CGO51591.2021.9370330

[63] Paul N. Whatmough, Chuteng Zhou, Patrick Hansen, Shreyas K. Venkataramanaiah, Jae-sun Seo, and Matthew Mattina. 2019. FixyNN: Energy-Efficient Real-Time Mobile Computer Vision Hardware Acceleration via Transfer Learning. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*, Ameet Talwalkar, Virginia Smith, and Matei Zaharia (Eds.). mlsys.org. https://proceedings.mlsys.org/book/281.pdf

[64] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. 2019. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*, David Z. Pan (Ed.). ACM, 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942149

[65] Yannan Nellie Wu, Vivienne Sze, and Joel S. Emer. 2020. An Architecture-Level Energy and Area Estimator for Processing-In-Memory Accelerator Designs. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 23-25, 2020*. IEEE, 116–118. https://doi.org/10.1109/ISPASS48437.2020.00024

[66] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. 2022. Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling. In *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*. IEEE, 1377–1395. https://doi.org/10.1109/MICRO56248.2022.00096

[67] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. 2021. HASCO: Towards Agile HArdware and Software CO-design for Tensor Computation. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 1055–1068. https://doi.org/10.1109/ISCA52012.2021.00086

[68] Tien-Ju Yang, Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2017. A method to estimate the energy consumption of deep neural networks. In *51st Asilomar Conference on Signals, Systems, and Computers, ACSSC 2017, Pacific Grove, CA, USA, October 29 - November 1, 2017*, Michael B. Matthews (Ed.). IEEE, 1916–1920. https://doi.org/10.1109/ACSSC.2017.8335698

[69] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 369–383. https://doi.org/10.1145/3373376.3378514

[70] Yifan Yang, Joel S. Emer, and Daniel Sánchez. 2023. ISOSceles: Accelerating Sparse CNNs through Inter-Layer Pipelining. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*. IEEE, 598–610. https://doi.org/10.1109/HPCA56546.2023.10071080

[71] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sánchez. 2021. Gamma: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 687–701. https://doi.org/10.1145/3445814.3446702

[72] Xinyi Zhang, Cong Hao, Peipei Zhou, Alex K. Jones, and Jingtong Hu. 2022. H2H: heterogeneous model to heterogeneous system mapping with computation and communication awareness. In *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, Rob Oshana (Ed.). ACM, 601–606. https://doi.org/10.1145/3489517.3530509

[73] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2017. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. *CoRR* abs/1707.01083 (2017). arXiv:1707.01083 http://arxiv.org/abs/1707.01083

[74] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. 2021. SARA: Scaling a Reconfigurable Dataflow Accelerator. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 1041–1054. https://doi.org/10.1109/ISCA52012.2021.00085

[75] Yang Zhao, Chaojian Li, Yue Wang, Pengfei Xu, Yongan Zhang, and Yingyan Lin. 2020. DNN-Chip Predictor: An Analytical Performance Predictor for DNN Accelerators with Various Dataflows and Hardware Architectures. In *2020 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2020, Barcelona, Spain, May 4-8, 2020*. IEEE, 1593–1597. https://doi.org/10.1109/ICASSP40776.2020.9053977

[76] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

[77] Size Zheng, Renze Chen, Yicheng Jin, Anjiang Wei, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2021. NeoFlow: A Flexible Framework for Enabling Efficient Compilation for High Performance DNN Training. *IEEE Transactions on Parallel and Distributed Systems* (2021).

[78] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 874–887. https://doi.org/10.1145/3470496.3527440

[79] Size Zheng, Siyuan Chen, Peidi Song, Renze Chen, Xiuhong Li, Shengen Yan, Dahua Lin, Jingwen Leng, and Yun Liang. 2023. Chimera: An Analytical Optimizing Framework for Effective Compute-intensive Operators Fusion. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*. IEEE.

[80] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020 [ASPLOS 2020 was canceled because of COVID-19]*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 859–873. https://doi.org/10.1145/3373376.3378508

[81] Shixuan Zheng, Xianjue Zhang, Leibo Liu, Shaojun Wei, and Shouyi Yin. 2022. Atomic Dataflow based Graph-Level Workload Orchestration for Scalable DNN Accelerators. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*. IEEE, 475–489. https://doi.org/10.1109/HPCA53966.2022.00042

[82] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. 2022. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 359–373. https://doi.org/10.1145/3503222.3507723

[83] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. 2017. Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*. IEEE Computer Society, 2242–2251. https://doi.org/10.1109/ICCV.2017.244

A ARTIFACT APPENDIX

A.1 Abstract

In this section, we provide detailed information that will facilitate the artifact evaluation process of TileFlow. TileFlow facilitates the analysis, design, and evaluation of complex dataflow architectures, making it a useful tool for researchers and engineers in the field of machine learning and hardware design. For Artifact Evaluation, we provide convenient scripts to reproduce the key experiments in the paper, as well as a tutorial on using TileFlow. In the following, we summarize the requirements and instructions to reproduce the experiments and play with TileFlow.

A.2 Artifact check-list (meta-information)

- **Algorithm:** TileFlow is a simulation framework to automate the analysis and design of fusion dataflow. The neural network and hardware are first described by users in TileFlow's tile-centric notation. Based on the notation, TileFlow characterizes the 3D design space of potential dataflow designs and evaluates each design point by the tree-based analysis. On top of that, TileFlow is able to search for high quality dataflow designs by exploring the design space using a combination of generative and Mont Caro Tree Search (MCTS).
- **Program:** Python (>= 3.8) and C++
- **Compilation:** Software construction tool 'scons' (v3.1.2) is used for validation experiment and the tutorials. 'cmake' (>= 3.12) is used for the dataflow comparison experiment.
- **Transformations:** No.
- **Binary:** No.
- **Model:** No.
- **Data set:** No.

- **Run-time environment:** Ubuntu 20.04.5 Linux system.
- **Hardware:** No.
- **Run-time state:** No.
- **Execution:** No.
- **Metrics:** Cycle.
- **Output:** The resulting figures shown in paper for key experiments.
- **Experiments:** One experiment for the validation of TileFlow's accuracy, and the other experiment for the comparison of different dataflows.
- **How much disk space required (approximately)?:** More than 4GB.
- **How much time is needed to prepare workflow (approximately)?:** Less than an hour.
- **How much time is needed to complete experiments (approximately)?:** The validation experiment completes in minutes. The dataflow comparison experiments take 1-2 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT.
- **Data licenses (if publicly available)?:** No data.
- **Workflow framework used?:** No.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.8350955

A.3 Description

We provide two software repositories for evaluation. The first is TileFlow main repository, and the second is Domino compiler. The main part of the method proposed in this paper is implemented in TileFlow repository. We develop Domino to provide a Python interface for TileFlow so that we can do experiments easily. It is not a critical part of TileFlow because TileFlow also has a programming interface using configuration files (.yaml format).

A.3.1 How to access. For the validation experiment and tutorials, please access the framework from
https://github.com/pku-liang/TileFlow.
For the dataflow comparison experiment, please access the source code from
https://github.com/KnowingNothing/Domino.

A.3.2 Hardware dependencies. The experiments only use CPU for simulation/modeling.

A.3.3 Software dependencies. The scripts need to run on linux systems and works best with sudo access. Make sure to install Python ≥ 3.8, scons ≥ v3.1.2, cmake ≥ 3.12.

A.3.4 Data sets. No data sets required.

A.3.5 Models. No models required.

A.4 Installation

Please follow the README file in each repository to install the software. We also list the brief instructions here. For a neat environment, please use a virtual environment for Python (e.g., conda or virtualenv).

Install dependencies (needs sudo).

```
$ sudo apt install scons libconfig++-dev
       libboost-dev libboost-iostreams-dev
       libboost-serialization-dev
       libyaml-cpp-dev libncurses-dev
       libtinfo-dev libgpm-dev
       git build-essential python3-pip
```

Download TileFlow and Domino.

```
$ cd ~
$ git clone --recursive \
    https://github.com/pku-liang/TileFlow.git
$ git clone --recursive \
    https://github.com/KnowingNothing/Domino.git
```

Build TileFlow.

```
$ cd ~/TileFlow
$ export TILEFLOW_BASE=$(pwd)
# build timeloop
$ cd 3rdparty/timeloop/src
$ ln -s ../pat-public/src/pat .
$ cd ..
$ scons -j4 --static
# build tileflow
cd ../..
scons -j4 --static
# required each time before using TileFlow
source ./setup-env.sh
```

Build Domino.

```
$ cd ~/Domino
$ mkdir build && cd build
$ cmake .. && make
$ cd ..
$ pip install -r requirements.txt
# required each time before using Domino
source ./set-env.sh
# use the Python interface for TileFlow
cd testing/tileflow
source ./set-env.sh
```

A.5 Experiment workflow

- **Experiment setup.** Download *TileFlow* and *Domino* and compile them according to the instructions in their README file (or see the instructions above).
- **The validation experiment.** This experiment validates TileFlow's accuracy with current simulator and a real hardware accelerator. Follow the instructions in
https://github.com/pku-liang/TileFlow/tree/master/AE/validation/timeloop
and
https://github.com/pku-liang/TileFlow/tree/master/AE/validation/accelerator
of *TileFlow*'s repository to reproduce experiment in Figure 7.
- **The dataflow comparison experiment.** This experiment compares different dataflow designs for the self-attention block and convolution neural networks on different hardware accelerator configurations. Please follow the instructions in
https://github.com/KnowingNothing/Domino/tree/master/testing/tileflow/test/experiments/
to reproduce the experiments in Figure 9/10.
- **Tutorials on using TileFlow.** In order to help users quickly get started with designing dataflows using TileFlow, we provide some tutorials. Please refer to the *tutorials* folder for instructions.

A.6 Evaluation and expected results

If run successfully, you will be able to see the experimental results in Figures 8, 10, and 11. The experimental results should generally be consistent with the results presented in the paper, but due to the randomness of the search process, some deviations might occur. Also, The image format in the experimental results would be slightly different from the format in the paper.

Received April 2023; revised August 2023; accepted September 2023