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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MICRO ’23, October 28–November 01, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-8-4007-0329-4/23/10...
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...
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 com-
puting 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 ac-
celerate 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 ex-
ample, Fused-Layer [2] proposes a tile-stationary dataflow to fuse
correlation 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 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 op-
erations 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. Chang-
ing 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 differ-
ent 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 com-
putation latency under hardware resource constraints. Manually
exploring the tiling space is impossible.

To design high-performance dataflows, performance models are
basic. However, state-of-the-art performance models [31, 38, 45]
focus on single operator acceleration. These models fail to provide
flexible and accurate program prediction for fusion dataflows. On one hand, these models treat the computation of a single opera-
tor as a polyhedron of iterations and formulate the performance prediction problem as calculating a polynomial composed of archi-
tecture 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 con-
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 bind-
ing, 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 perform-
ance of fusion dataflows for general DNN layers and customized
accelerator architectures. We also make TileFlow open-source on
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 per-
formance critical metrics: data movement volume and re-
source usage.

• We implement the proposed modeling techniques into a
framework called TileFlow to help dataflow/mapping evalu-
ation, 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. Tile-
Flow 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
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 x 16 x 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 (L1 - LN buffers) 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 x K, L = Softmax(S), A = V x 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 x 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 (Op1) into another operator (Op2) is to insert the iteration space of Op1 into the iteration space Op2. 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 unwanted 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 CNN [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.
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.

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 Op1 is fused into another operator Op2, Op1’s iteration space (a polyhedron) is inserted into the iteration space of Op2, which is modeled as inserting a node (for Op1) as the child to another node (for Op2) 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 Op1 into Op2), only the reduction loops of Op2 are allowed in the parent tile in the result ordering tree (as shown in Figure 4 part c). Otherwise, if the reduction loops of Op1 appear in parent tile (as outer loops), Op2 can’t start execution until Op1 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbr.</th>
<th>Explanation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial</td>
<td>Sp</td>
<td>map loops to spatial units</td>
<td>Sp(loops)</td>
</tr>
<tr>
<td>Temporal</td>
<td>Tp</td>
<td>map loops to temporal steps</td>
<td>Tp(loops)</td>
</tr>
<tr>
<td>Pipeline</td>
<td>Pipe</td>
<td>tiles are dependent and execute in a pipeline manner</td>
<td>Pipe(Τ₁, ..., Τ_M)</td>
</tr>
</tbody>
</table>
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 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 propose the fusion dataflow design choices in the 3D space, we use our tile-centric notation to express a fusion dataflow. In the notation, a tile $(T_n)$ at memory level $n$ is defined as

$$T_n = \{T_{11}^n, T_{12}^n, \ldots \} = \{T_{21}^{n-1}, T_{22}^{n-1}, \ldots \}$$

where $\{T_{11}^n, T_{12}^n, \ldots \}$ is a loop nest over a list of sub-tiles $(T_{11}^{n-1}, T_{12}^{n-1}, \ldots)$, 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:** $T_0^0 = \{i_0, b_0, k\} (A)$, $T_0^1 = \{i_0, b_0, k\} (B)$, $T_0^2 = \{i_0, b_0, k\} (C)$
- **level 1:** $T_1^0 = \{i_1, l_1\} (T_0^1)$, $T_1^1 = \{i_1, l_1\} (T_0^1)$
- **level 2:** $T_2^0 = \{i_2, j_2, l_2\} (T_1^1)$

**Inter-tile (Binding):**

- $Pipe(T_0^0, T_0^1)$, $Shar(T_0^0, T_0^1)$, $Sp(i_2)$, $Sp(j_2)$, $Sp(l_2)$

This dataflow uses the same ordering in Figure 4 part c). For inter-tile, $T_0^0, T_0^1, T_1^0, T_1^1$ 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 $i_0, i_1, l_2$ 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 update this as:

\[
\text{Slice}_{Z}^t = [b_i^t : e_i^t, b_i^{t+1} : e_i^{t+1}]
\]

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 \( 4 \times 6 \).

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:

\[
DM_Z^t = |\text{Slice}_{Z}^t - \text{Slice}_{Z}^{t-1}|
\]

Note that \( \text{Slice}_{Z}^t \) is a set of data, so \( \text{Slice}_{Z}^t - \text{Slice}_{Z}^{t-1} \) 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 normal 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:

\[
DM_Z = \sum_{t_i \in \text{Bounds}} U_i
\]

where:

\[
U_i = \begin{cases} 
|U_i| \times (DM_Z^{t_i} + (|U_{i-1}| - 1) \times U_{i-1}), & i > 1 \\
DM_Z^{t_0}, & i = 0 
\end{cases}
\]

where \( \text{Bounds} \) is the set of step time step boundaries \( t_i (0 \leq i \leq Dim, \text{Dim} \) 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 \( t_1, t_2 \), so the time vector has two values). Tensor \( A \) needs \( 4 \times 4 \) new elements for the next step computation and reuses \( 4 \times 2 \) elements from previous step, tensor \( B \) needs \( 4 \times 3 \) elements, while tensor \( C \) is fully reused and no data movement is needed. We also show the time step boundaries in Figure 5 \( (t_0 = [0,0], t_1 = [0,2], \text{and } t_2 = [2,0]) \). 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).

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 \( \text{Shar}, \text{Para}, \) and \( \text{Pipe} \), the data movement calculation approach is the same as above. But for \( \text{Seq} \), 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 $T_{i-1}^1, T_{i-1}^2$, the number of PE used by the parent tile depends on the inter-tile resource binding primitive. The calculation formula is

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

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

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

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

$$\text{Lat}_{T_n} = \begin{cases} \text{Perfect_Tile_Latency}(T_n, \text{Spec}), & \text{if } T_n \text{ has no sub-tiles} \\ \max(\frac{DM_{\text{load}}}{BW_n}, \sum_i \{\text{Lat}_{T_{i-1}^1}, \frac{DM_{\text{store}}}{BW_n}\}), & \text{Seq or Shar} \\ \max(\frac{DM_{\text{load}}}{BW_n}, \max_i \{\text{Lat}_{T_{i-1}^1}, \frac{DM_{\text{store}}}{BW_n}\}), & \text{otherwise} \end{cases}$$

where $\text{Perfect_Tile_Latency}$ 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 $\text{mem}$ represents which level of memory to stage intermediate data; $\text{target}$ represents which operator to fuse into; $\text{binding}$ represents the binding primitive. In the example, $OP_1$ is fused to $OP_3$ at memory $L1$ and $OP_2$ is fused to $OP_4$ 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 7.84m² 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.

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
Fuse two convolutions with.

Batch, multi_heads, and row dimension. Pipeline batch and multi_heads dimension.

Height and width dimensions tiled.

Normalized Performance

(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.

![Graphs showing normalized performance](image)

Figure 9: Performance of TileFlow Mapper

Table 5: Different dataflows in evaluation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layerwise</td>
<td>No fusion. Map one Op to hardware at a time.</td>
</tr>
<tr>
<td>Uni-pipe</td>
<td>Pipeline $Q \times K$ and softmax without tiling multi_heads/row.</td>
</tr>
<tr>
<td>FLAT-HGran  [26]</td>
<td>Fuse $Q \times K$ and softmax and tile batch, multi_heads dimension.</td>
</tr>
<tr>
<td>FLAT-RGran  [26]</td>
<td>Fuse $Q \times K$ and softmax and tile batch, multi_heads, and row dimension.</td>
</tr>
<tr>
<td>Chimera     [79]</td>
<td>Fuse $Q \times K$ and softmax and tile all the dimensions.</td>
</tr>
</tbody>
</table>

Convolution Chain Dataflow

Layerwise | No fusion. Map one convolution to hardware at at time. |
ISOS [70] | Fuse two convolutions with only width tiled. |

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 6.65x. Compared to the best self-attention dataflow (FLAT-RGran) in Table 5, the average speedup is 1.85x. 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 1.31x. The speedup to Fused-Layer is 1.28x.

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.
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 \textit{Layerwise}, \textit{Uni-pipe} dataflow achieves 1.62x speedup, \textit{FLAT-HGran} dataflow achieves 3.59x speedup, \textit{FLAT-RGran} dataflow achieves 2.89x speedup, and \textit{Chimera} dataflow achieves 2.91x speedup. \textit{TileFlow} achieves the best performance compared to all the other dataflows: 6.65x speedup to \textit{Layerwise}; 4.11x speedup to \textit{Uni-Pipe}; 1.85x speedup to \textit{FLAT-HGran}; 2.30x speedup to \textit{FLAT-RGran}; 2.28x speedup to \textit{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 \textit{Layerwise} dataflow, \textit{Uni-pipe} and \textit{FLAT-HGran} dataflows can reduce 90.4% DRAM access, \textit{FLAT-RGran} dataflow can reduce 81.5% DRAM access, \textit{Chimera} dataflow can reduce 75.1% DRAM access, \textit{TileFlow} dataflow can reduce 87.1% DRAM access. The reduction of DRAM access implies a higher data reuse on on-chip memory. Although \textit{FLAT-HGran} has less DRAM data movement than \textit{TileFlow} for some of the input configurations, its PE utilization is not high (about 50% of \textit{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 \times -6.45\times 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, \textit{Uni-pipe} is better than \textit{Layerwise} dataflow because the fusion eliminates a large number of DRAM access; \textit{FLAT-HGran} dataflow is better than \textit{Uni-pipe} dataflow because of tiling, the tiled blocks are spatially mapped to different cores and increase parallelism; \textit{FLAT-RGran} and \textit{Chimera} dataflows produce similar performance to \textit{FLAT-HGran}, but their L1 buffer footprint is smaller; \textit{FLAT-RGran} only requires 28.4% of the L1 buffer size that \textit{FLAT-HGran} uses for computation, while \textit{Chimera} only requires 14.8%.
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 effect 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 12000GB/s by step 1GB/s. The metric we use for bandwidth is slow-down [45]:

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

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.

<table>
<thead>
<tr>
<th>PE Size</th>
<th>8×</th>
<th>16×</th>
<th>32×</th>
<th>64×</th>
<th>128×</th>
<th>256×</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline</td>
<td>12.58</td>
<td>3.15</td>
<td>2.36</td>
<td>1.73</td>
<td>1.57</td>
<td>1.57</td>
</tr>
<tr>
<td>TileFlow</td>
<td>6.29</td>
<td>1.57</td>
<td>1.57</td>
<td>1.57</td>
<td>1.57</td>
<td>1.57</td>
</tr>
</tbody>
</table>

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

**Part a) No Tiling Exploration, No Memory Limit.**

<table>
<thead>
<tr>
<th>Dataflow</th>
<th>MGran</th>
<th>BGran</th>
<th>HGran</th>
<th>RGran</th>
<th>TileFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle (10^6)</td>
<td>335.34</td>
<td>151.00</td>
<td>67.11</td>
<td>18.87</td>
<td>16.78</td>
</tr>
<tr>
<td>L1 Used (MB)</td>
<td>131.14</td>
<td>4.10</td>
<td>0.26</td>
<td>0.17</td>
<td>0.01</td>
</tr>
<tr>
<td>L2 Used (MB)</td>
<td>262.14</td>
<td>16.38</td>
<td>4.10</td>
<td>2.18</td>
<td>0.19</td>
</tr>
</tbody>
</table>

**Part b) With Exploration for Tiling, No Memory Limit.**

<table>
<thead>
<tr>
<th>Dataflow</th>
<th>MGran</th>
<th>BGran</th>
<th>HGran</th>
<th>RGran</th>
<th>TileFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle (10^6)</td>
<td>335.34</td>
<td>8.39</td>
<td>8.39</td>
<td>8.39</td>
<td>7.67</td>
</tr>
<tr>
<td>L1 Used (MB)</td>
<td>131.14</td>
<td>65.57</td>
<td>8.20</td>
<td>2.11</td>
<td>0.16</td>
</tr>
<tr>
<td>L2 Used (MB)</td>
<td>262.14</td>
<td>131.07</td>
<td>131.07</td>
<td>196.61</td>
<td>131.07</td>
</tr>
</tbody>
</table>

**Part c) With Exploration for Tiling, With Memory Limit.**

<table>
<thead>
<tr>
<th>Dataflow</th>
<th>MGran</th>
<th>BGran</th>
<th>HGran</th>
<th>RGran</th>
<th>TileFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle (10^6)</td>
<td>OOM</td>
<td>OOM</td>
<td>14.68</td>
<td>14.68</td>
<td>16.78</td>
</tr>
<tr>
<td>L1 Used (MB)</td>
<td>-</td>
<td>-</td>
<td>4.10</td>
<td>0.53</td>
<td>0.05</td>
</tr>
<tr>
<td>L2 Used (MB)</td>
<td>-</td>
<td>-</td>
<td>32.77</td>
<td>12.29</td>
<td>20.48</td>
</tr>
</tbody>
</table>

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-BGran. 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 ($S = Q \times K, L = \text{Softmax}(S), A = V \times L$) 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 $K, L, V$, 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 example. 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.
These accelerators employ different dataflows for DNN acceleration. 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 datatflow 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 polygonal 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 polyhedral-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.

### 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 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 1.85X runtime speedup compared to state-of-the-art work on average for self-attention and 1.28X 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


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:** 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 Monte Carlo 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.
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 libibm-dev
   libgmp-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

# required each time before using Domino
$ 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