SmartMem: Layout Transformation Elimination and Adaptation for Efficient DNN Execution on Mobile

Wei Niu
wniu@uga.edu
University of Georgia
Athens, GA, USA

Md Musfiqur Rahman Sanim
musfiqur.sanim@uga.edu
University of Georgia
Athens, GA, USA

Zhihao Shu
Zhihao.Shu@uga.edu
University of Georgia
Athens, GA, USA

Jiexiong Guan
jguan@wm.edu
William & Mary
Williamsburg, VA, USA

Xipeng Shen
xshen5@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Miao Yin
miao.yin@uta.edu
University of Texas at Arlington
Arlington, TX, USA

Gagan Agrawal
gagrawal@uga.edu
University of Georgia
Athens, GA, USA

Bin Ren
bren@wm.edu
William & Mary
Williamsburg, VA, USA

Abstract

This work is motivated by recent developments in Deep Neural Networks, particularly the Transformer architectures underlying applications such as ChatGPT, and the need for performing inference on mobile devices. Focusing on emerging transformers (specifically the ones with computationally efficient Swin-like architectures) and large models (e.g., Stable Diffusion and LLMs) based on transformers, we observe that layout transformations between the computational operators cause a significant slowdown in these applications. This paper presents SmartMem, a comprehensive framework for eliminating most layout transformations, with the idea that multiple operators can use the same tensor layout through careful choice of layout and implementation of operations. Our approach is based on classifying the operators into four groups, and considering combinations of producer-consumer edges between the operators. We develop a set of methods for searching such layouts. Another component of our work is developing efficient memory layouts for 2.5 dimensional memory commonly seen in mobile devices. Our experimental results show that SmartMem outperforms 5 state-of-the-art DNN execution frameworks on mobile devices across 18 varied neural networks, including CNNs, Transformers with both local and global attention, as well as LLMs. In particular, compared to DNNFusion, SmartMem achieves an average speedup of 2.8×, and outperforms TVM and MNN with speedups of 6.9× and 7.9×, respectively, on average.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04.
https://doi.org/10.1145/3620666.3651384

ACM Reference Format:
Wei Niu, Md Musfiqur Rahman Sanim, Zhihao Shu, Jiexiong Guan, Xipeng Shen, Miao Yin, Gagan Agrawal, and Bin Ren. 2024. SmartMem: Layout Transformation Elimination and Adaptation for Efficient DNN Execution on Mobile. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620666.3651384

1 Introduction

As Machine Learning (ML), more specifically, Deep Learning (DL) and Deep Neural Networks (DNNs) have permeated our every day life, there is a growing need for supporting inference using DL models on the ubiquitous mobile devices. From the application side, possibility are endless and go well beyond common speech or image recognition. On the other hand, we have the growing computational capacity (and continued popularity) of smartphones.

For the past several years, inference with even fairly complex models has been feasible on mobile devices [10, 11, 23, 31, 32, 60]. Such inference has the benefit that a user's data does not need to be transmitted to a cloud or a server. This, in turn, allows ML models to execute when the device is not connected to the internet, and alleviates privacy concerns about sharing personal information that many users frequently have [16, 48].

Recently, Transformers [76] have revolutionized the fields of computer vision (CV) [3, 5, 19, 20, 46] and natural language processing (NLP) [4, 17, 21, 57, 65, 67]. Transformer-based models uniquely provide long-range dependency handling and global contextual awareness, which are driving existing popular AI applications such as ChatGPT, Bard, and Alexa.

Table 1. Latency and transformation breakdown across various models. 'Lat.' shorts for Latency. 'Exp.' refers to the latency associated with explicitly transforming the tensor's layout, such as `Transpose` and `Reshape`. 'Imp.' denotes the latency incurred by implicit layout transformations. 'Comp.' indicates the latency attributed to the remaining operators. 'SD' represents Stable Diffusion model. These results are collected using MNN [32] on a Snapdragon 8 Gen 2 platform. MACs means the number of multiply-accumulate operations, and GMACS represents giga MACs per second.

Model	#MACs (G)	#Layout transform	Lat. (ms)	Lat. breakdown (%) Imp.	Exp.	Comp.	Speed (GMACS)
ResNet50 [27]	4.1	3	14	4.8	0.2	95	**293**
FST [34]	162	32	1,506	70.7	1.8	27.5	**108**
RegNet [58]	3.2	6	57	16.7	0	83.3	**56**
CrossFormer [69]	5.0	208	336	15.3	55.2	29.5	**15**
Swin [46]	4.6	242	342	14.7	54.1	31.2	**15.2**
AutoFormer [8, 9]	4.7	233	335	13.3	54.2	32.5	**14**
CSwin [19]	6.9	769	703	14.3	50.2	35.5	**10**
SD-TextEncoder [59]	6.7	183	133	15.1	36.3	48.6	**44**
SD-UNet [59]	90	533	2172	19.4	42.1	38.5	**42**
Pythia-1B [6]	119	385	3034	11.7	31.7	56.6	**39**

Studies assessing them as computational workloads [36] have identified that as compared to the CNN-based designs, the computation graph representations for transformers are more complex, specifically, they have more data flow splits, shuffles, and merges. One further development has been the emergence and popularity of (computationally) efficient local-attention (Swin-like) [9, 46, 69, 77] Transformers that have reduced computational complexity, though at the cost of more layout transformations.

Table 1 summarizes this aspect. Specifically, three of the older ConvNets (like ResNet50 [27]), six newer (Transformer) models, and one decoder-only LLM (Pythia [6]) are compared with respect to the time spent on implicit and explicit data layout transformations (as compared to pure computations). A majority of the older models spend a relatively small fraction of their time on (implicit or explicit) layout transformations. On the other hand, Transformer models all consistently spend between 43% to 70% of their time on data transformation. Moreover, the execution speed of these models is, on the average, around one order of magnitude slower than earlier models. It seems likely that increased numbers of data transformation (as indicated in the third column of Table 1) cause poor locality during compute-oriented operations, resulting in a significant slowdown.

In this work, we take the position that almost all *layout transformations* can be eliminated and instead, the layout of the tensor that is produced can be chosen to serve various computational operations efficiently. This paper presents a systematic framework for enabling elimination of such unnecessary memory intensive operations. Components of the work include the following:

- Careful study of the relationship between the computation and input/output data layout of DNN operators, and a high-level operator type classification based on operators' performance sensitivity to input layout and the output layout customizability.
- A procedure for layout transformation elimination and an effective heuristic method for selecting the layout for each operator that is not fused or eliminated.
- A procedure to map the chosen layout to memory hierarchy by taking 2.5D (texture) memory into consideration.
- Integration of the above into a comprehensive framework called `SmartMem`, which is then implemented on top of a state-of-the-art end-to-end DNN execution framework DNNFusion [51].

`SmartMem` has been extensively evaluated on 18 cutting-edge DNNs, including 4 ConvNet models, 6 Transformer models, and 8 Hybrid (combining ConvNet and Transformer structures) models on mobile GPUs. The evaluation demonstrates a significant speedup compared to 5 state-of-the-art DNN execution frameworks (MNN [32], NCNN [50], TFLite [1], TVM [10], and DNNFusion [51]). `SmartMem` reduces the number of operators by 21% to 65% compared with other frameworks. In terms of latency, `SmartMem` achieves an average speedup of 2.8× over DNNFusion, a state-of-the-art baseline. Compared with two other popular frameworks, TVM and MNN, `SmartMem` achieves an average speedup of 6.9× and 7.9×, respectively. Furthermore, `SmartMem` enhances cache utilization and reduces memory pressure, enabling the execution of some models on resource-constrained devices while other frameworks may encounter challenges.

2 Background and Motivation

2.1 DNN Recent Advances: Transformers.

The Transformer architecture has become the dominant paradigm in deep learning space, leveraging *attention mechanism* to focus on different parts of the input sequence while generating each part of the output sequence [4, 17, 57, 67]. One notable challenge with the standard (or global) attention mechanism is its computational and space complexities, both of which are $O(n^2)$, where n represents the length of the input sequence. DL practitioners have built upon the foundational (vanilla) Transformer model by introducing local-attention Transformer models [8, 9, 18, 43, 46] that reduce computational complexity.

Local attention focuses on a subset of input tokens (typically within a window) at a time and requires less computation. To achieve computational reduction, frequent explicit shape/data reorganization (Reshaping, Transposing, Gathering) is used to split and transpose segments within the input data into these smaller windows. Another trend involves combining traditional ConvNets with Transformers and designing new model structures [7, 12, 19, 25, 68, 74, 77] to benefit from both structures. These new structures introduce

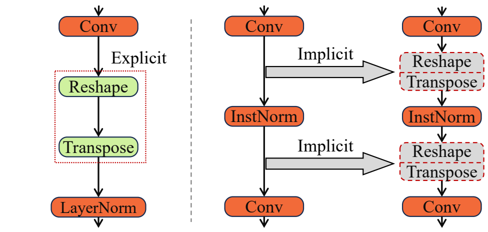


Figure 1. Examples of layout transformation in DNNs.

implicit data reorganization due to different layout preferences for various operators. Figure 1 shows the two types of data reorganization, which we will discuss next.

2.2 Motivation and Research Issues

To motivate the key optimizations we are proposing, we further discuss two examples from Figure 1. In sub-figure (a), we show two computational operations, a `Conv` (convolution) and a `LayerNorm`. In between these two, the programmer explicitly inserts two operations, a `Reshape` and a `Transpose`. A reshape operation changes the number and sizes of dimensions for the tensor – for example, a $5 \times 5 \times 5$ 3-D tensor can be recast as a 5×25 2-D matrix. A subsequent transpose operation can next convert it to a 25×5 matrix.

In Figure 1 (a), the `Reshape` and `Transpose` are considered *explicit*, in the sense they are inserted explicitly by a model implementer while working with a framework such as MNN [32]. In Figure 1 (b) we show what we consider as an *implicit* layout transformation. In this example, a `Conv` operation is followed by an `InstNorm` or Instance Normalization operation, which is then followed by another `Conv`. A framework such as MNN inserts both `Reshape` and `Transpose` examples before and after the `InstNorm`, aligning them with predefined input layouts.

The underlying reason behind these operations is that different operators in a DNN might require input tensors with different shapes and/or layouts. For instance, fully connected operators require a *flattened input*, whereas convolutional operators require multi-dimensional inputs to perform spatial operations. A transpose operation consumes high memory bandwidth, and furthermore, can reduce locality for the operations that follow the transpose. Eliminating reshape and transpose operations, and performing the two operations, i.e., those before and after the transpose, on the same layout should be able to improve performance. However, eliminating such layout transformation involves many important (inter-dependent) questions.

- **Q1:** In a large and complex computational graph capturing a large DNN, deciding when two consecutive operations should use the same layout of data.
- **Q2:** For two operations where we decide to use the same layout, choosing the layout that leads to efficient execution of both operands.

Table 2. Memory comparison on mobile GPUs.

	Characteristics	1D Buffer	**2.5D Texture**
Computation	Acceleration engine	N	**Y**
	Automatic bounds checking	N	**Y**
	Hardware interpolation	N	**Y**
Data access	Organization	Contiguous	**Multidimensional**
	Addressing	Pointer-based	**Coordinates**
	Dedicated cache	No	**Yes**
	Data locality	1D	**2.5D**
	Direct memory access on CPU	Yes	**No**

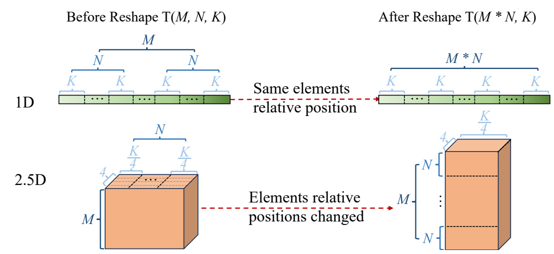


Figure 2. Layout transformation and 2.5D memory.

- **Q3:** Implementing an operation efficiently for a chosen layout (distinct from the original layout), including deciding access pattern and simplifying index computations.
- **Q4:** Mapping the chosen layout to memory hierarchy, especially the 2.5D memory (further discussed in Section 2.3).

Many of these issues have been addressed previously in computer systems and compiler community, though not specific to our target workload. The closest work in this area is on integrating data layout selection and loop transformations [22, 35, 49, 52, 61, 62]. However, the most significant difference in our work is that prior efforts, motivated by traditional scientific workloads, have focused on one nested loop. In comparison, with transformers (or even other DNNs), the challenge is making this selection among a Directed Acyclic Graph (DAG) of operators (i.e. the computational graph). Other differences in the work involve deciding when eliminating a transpose is beneficial or not, implementing multiple operators efficiently on the same layout, and physically mapping to 2.5D memory.

2.3 GPU texture memory

The GPU texture memory specializes in improving two-dimensional spatial locality. Since each cache element can be a vector with a fixed length of four, and the cache itself has two dimensions (referred to as width and height), we refer it to as 2.5D memory. Originally designed to facilitate graphical rendering processes, this design also offers significant advantages for stencil and similar computations. For instance, using 2.5D texture memory for convolution operations can result in a $3.5\times$ reduction in latency compared to 1D buffer memory [31]. Table 2 summarizes the main differences between GPU texture memory and 1D buffer memory.

Table 3. Operator classification based on input layout dependency and output layout flexibility. 'Comp.' stands for Computation.

Comp. performance \ Output layout	Variable (Computation order dependent)	Fixed
Input layout dependent (ILD)	Conv: $C^{l}_{4\text{-}d} \leftarrow A^{l_1}_{4\text{-}d} * B^{l_2}_{4\text{-}d}$ MatMul: $C^{l}_{2\text{-}d} \leftarrow A^{l_1}_{2\text{-}d} \cdot B^{l_2}_{2\text{-}d}$ LayerNorm: $C^{l}_{n\text{-}d} \leftarrow A^{l_1}_{n\text{-}d} \odot B^{l_2}_{2\text{-}d}$ Softmax: $C^{l}_{n\text{-}d} \leftarrow A^{l_1}_{n\text{-}d} \odot B^{l_2}_{2\text{-}d}$	Reshape: $B^{L}_{n\text{-}d} \leftarrow A^{l_1}_{m\text{-}d}$ Transpose: $B^{L}_{n\text{-}d} \leftarrow A^{l_1}_{n\text{-}d}$ DtoS: $B^{L}_{n\text{-}d} \leftarrow A^{l_1}_{n\text{-}d}$ StoD: $B^{L}_{n\text{-}d} \leftarrow A^{l_1}_{n\text{-}d}$
Input layout independent (ILI)	Unary: $B^{l}_{n\text{-}d} \leftarrow A^{l_1}_{n\text{-}d}$ Add: $C^{l}_{n\text{-}d} \leftarrow A^{l_1}_{n\text{-}d} + B^{l_1}_{n\text{-}d}$	Gather: $B^{L}_{n\text{-}d} \leftarrow A^{l_1}_{m\text{-}d}$ Slice: $B^{L}_{n\text{-}d} \leftarrow A^{l_1}_{n\text{-}d}$

Unary refers to an operator applying a function to each element of a single input. DtoS and StoD mean DepthToSpace and SpaceToDepth, respectively.

Coupled with the advantages associated with spatial locality, there are also some challenges. Figure 2 shows an example of `Reshape` in 1D and 2.5D memory. As a background, reshaping a tensor involves changing its shape without altering the underlying data order. For a 1D memory, due to the linear format, reshaping simply implies interpreting the same data with a different number of dimensions. The overhead is negligible as it only involves changing the metadata of the tensor. However, for 2.5D memory, because spatial relationships between data points are important (consider image processing or matrix computations), reshaping a tensor in 2.5D memory involves more complex processes of reordering the data layout while maintaining inherent relationships within the data. Furthermore, due to the limited bandwidth, the data transformation overhead is crucial when an operation requires moving both metadata and actual data (such as explicit and implicit data transpose) in 1D linear buffer and 2.5D texture memory on mobile GPUs.

3 Design of SmartMem

Our framework has three components, which address the four questions listed earlier in Section 2.2: 1) an operator type classification based on operators' performance sensitivity to input layout as well as the output layout customizability, and an analysis built on this operator type classification (to answer Q1), 2) a layout transformation elimination procedure and a method for selecting the optimal layout for each (not eliminated) operator based on the operator type classification and layout transformation analysis (to answer Q2 and Q3), and 3) a further optimization procedure that maps the chosen layout to memory hierarchy by taking 2.5D memory into account (to answer Q4).

3.1 Operator Classification and Analysis

The foundation of our method is a novel classification system for operators commonly seen in DNNs. Any given operator in our target workload is classified along two dimensions.

Table 4. Operator type definition.

Name	Definition
Input Layout Dependent & Variable	$B_{\sigma(i),\sigma(j),\ldots,\sigma(n)} = \mathcal{O}^{ILD-Variable}_{\pi(i),\pi(j),\ldots,\pi(m)}(A_{i,j,\ldots,m})$
Input Layout Dependent & Fixed	$B_{i,j,\ldots,n} = \mathcal{O}^{ILD-Fixed}_{\pi(i),\pi(j),\ldots,\pi(m)}(A_{i,j,\ldots,m})$
Input Layout Independent & Variable	$B_{\sigma(i),\sigma(j),\ldots,\sigma(n)} = \mathcal{O}^{ILI-Variable}(A_{i,j,\ldots,m})$
Input Layout Independent & Fixed	$B_{i,j,\ldots,n} = \mathcal{O}^{ILI-Fixed}(A_{i,j,\ldots,m})$

The first dimension is whether the performance of the computation depends upon the input layout or is independent. The second dimension is whether the output layout is customizable (perhaps in view of the computation pattern chosen for operator's implementation). These two dimensions result in four quadrants and each operator can be mapped to one quadrant. In some cases, one operator may be placed in different quadrants depending on whether the layout of its different operands is the same or different. Table 3 shows sample operators for each quadrant.

To explain these quadrants, we start at the bottom left and consider the addition operation:

$$C^{l}_{n\text{-}d} \leftarrow A^{l_1}_{n\text{-}d} + B^{l_1}_{n\text{-}d}$$

In the above operation, tensors A and B have the same layout denoted as l_1. Having an identical layout, and with an addition operation needing to touch each element once, the computational performance is not sensitive to the layout l_1, i.e., addition can be performed efficiently with traversal of the two matrices matching their (identical) physical layout. At the same time, the layout of the output tensor C can be customized, for example, based on the needs of the downstream operations.

Moving clockwise, we consider matrix multiplication example $C^{l}_{2\text{-}d} \leftarrow A^{l_1}_{2\text{-}d} \cdot B^{l_2}_{2\text{-}d}$. As a multiplication operation involves temporal reuse of data, the performance of the operation is clearly sensitive to the input layouts (even if the layouts l_1 and l_2 are identical). At the same time, the output layout can be customized for this application. An operator like `Transpose`, on the other hand, has a well-defined output layout (dependent upon the input layout). Given the memory transformations involved, the performance can be sensitive to the layout of the input operand. Finally, consider an operator like `Slice`. Because of simple selection involved, the performance is insensitive to the layout of the input. Moreover, by definition, the output layout has to match the input layout and is therefore not customizable

We next formally define these four quadrants (or types) of operators so that we can classify and place a given operator into this table. Table 4 shows the formal definition to these four quadrants (or types) of operators so that we can classify and place a given operator into Table 3.

Table 5. Operator combination - action. ILD & Var is short for ILD & Variable.

First \ Second		ILD & Variable	ILI & Variable	ILD & Fixed	ILI & Fixed
ILD & Var	Action	Keep both	Try fuse	Eliminate 2nd	Eliminate 2nd
ILI & Var	Action	Try fuse	Try fuse	Eliminate 2nd	Eliminate 2nd
ILD & Fixed	Action	Eliminate 1st	Eliminate 1st	Eliminate both	Eliminate both
ILI & Fixed	Action	Eliminate 1st	Eliminate 1st	Eliminate both	Eliminate both

Take Input Layout Dependent & Variable (ILD-Variable) as an example. Let A and B represent the input tensor and the output tensor, respectively, and $\mathcal{O}^{ILD-Variable}$ denotes the operator. The permutations π and σ represent the flexibility in processing and generating data, respectively. More specifically, The permutation π represents a rearrangement of these dimensions that affects the input layout, and σ represents a rearrangement of the computed result's dimensions affecting the output layout. Denoting mathematically,

$$B_{\sigma(i),\sigma(j),\ldots,\sigma(n)} = \mathcal{O}^{ILD-Variable}_{\pi(i),\pi(j),\ldots,\pi(m)}(A_{i,j,\ldots,m})$$

The input tensor is accessed as $A'_{\pi(i),\pi(j),\ldots,\pi(m)} = A_{i,j,\ldots,m}$ during the computation, where A' is the tensor A with its layout altered according to π. This rearrangement can lead to different memory access patterns, which might impact cache performance and processing speed. The computation is then performed, and the result is organized in the output tensor B as $B'_{i,j,\ldots,n} = \mathcal{O}(A'_{\pi(i),\pi(j),\ldots,\pi(m)})$. Finally, the output tensor B is produced with a layout determined by σ, i.e. $B_{\sigma(i),\sigma(j),\ldots,\sigma(n)} = B'_{i,j,\ldots,n}$.

Each of these categories reflects a different combination of data layout considerations and computational patterns, which are crucial for optimizing the performance of DNNs. The permutation functions π and σ accommodate the flexibility in data layout and output generation, which can be exploited for performance enhancements such as minimizing memory traffic, improving cache usage, or optimizing parallel execution strategies.

3.2 Layout Transformation Elimination Analysis

The aforementioned operator classification reveals the relationship between the computation and input/output layouts of an operator. As stated above, the four combinations are: input layout dependent and variable output (ILD & Variable), input layout independent and variable output (ILI & Variable), input layout dependent and fixed output (ILD & Fixed), and input layout independent and fixed output (ILI & Fixed). From a performance optimization perspective, we observe that their "optimization complexity" gradually decreases. For example, ILD & Variable requires us to be aware of both the input and output layouts while ILI & Fixed has no requirement about either of the layouts.

Based on this key insight, Table 5 summarizes SmartMem's computation optimizations (on a DNN computational graph) for each pair of DNN operators. Specifically, SmartMem has four levels of computation optimizations (marked with different colors): keeping both operators, trying to fuse them, eliminating one operator (either the first or the second), and eliminating both operators, which represent the optimization opportunities from low to high. Correspondingly, Table 6 summarizes the resulting output type and the input/output layout search policies after the computation optimizations explained above. The resulting output type is decided by the operator with a higher optimization complexity or the preserved operator, for example, after the computation optimization of a pair of operators with ILD & Variable and ILI & Variable types, respectively, the resulting (fused) operator is ILD & Variable, and after eliminating the second operator in a pair of ILD & Variable and ILD & Fixed operators, the resulting operator is in ILD & Variable, too.

Table 6. Operator combination and their corresponding design decisions.

First \ Second		ILD & Var	ILI & Var	ILD & Fixed	ILI & Fixed
ILD & Var	Output Layout	ILD & Variable* Search both	ILD & Variable Search fused	ILD & Variable Search 1st	ILD & Variable Search 1st
ILI & Var	Output Layout	ILD & Variable Search fused	ILI & Variable No search	ILI & Variable No search	ILI & Variable No search
ILD & Fixed	Output Layout	ILD & Variable Search 2nd	ILI & Variable No search	N/A No search	N/A No search
ILI & Fixed	Output Layout	ILD & Variable Search 2nd	ILI & Variable No search	N/A No search	N/A No search

* Both operators are ILD & Variable.

The input/output layout search also has four levels (marked with different colors): searching input and output layouts for both operators, searching input and output layouts for the fused operator, searching for either the first or the second, and no need to perform any search operation, representing the varied levels of optimization processing difficulties from high to low. It is worth noting that the layout search only happens for the operator pairs involving ILD & Variable. To explain the ideas, consider a pair of operators, i.e., `Conv+Reshape` (ILD & Variable + ILD & Fixed) as an example. Table 5 implies that `Reshape` can be eliminated[1]. According to Table 6, the preserved operator (`Conv`) is still in ILD & Variable and SmartMem needs to search for its input layout.

To achieve both computation and layout selection optimizations, SmartMem specifically answers these three questions:

- How to fuse operators? Specifically how to decide if an operator fusion is legal and profitable?
- How to correctly and effectively eliminate any operators (specifically, the layout transformation operators in the types of ILD & Fixed or ILI & Fixed)?
- How to select data layout for fused/preserved operators?

[1]The subtle difference between operator fusion and elimination will be elaborated in Section 3.2.1.

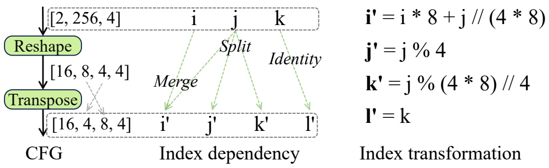


Figure 3. An example of index dependency and transformation. left: a computational graph comprising `Reshape` and `Transpose`, middle: index dependency and transformation from the input, right: index computation (no opt.).

With respect to the first question, `SmartMem` relies on the techniques based on the DNNFusion project [51] to decide if an operator fusion is legal. Based on DNNFusion, the subsequent operator elimination and layout selection designed in `SmartMem` bring more opportunities for beneficial fusions, enhancing execution performance (as we show through our evaluation results in Section 4). Thus, this section mainly focuses on the new operator elimination and layout selection techniques to answer the last two questions.

3.2.1 Operator Elimination based on Index Comprehension. All cases (except the first marked by red) in Table 5 can be optimized by an operator fusion (by following the rules in DNNFusion [51]). Going beyond operator fusion, it turns out that a more advanced optimization called *Operator Elimination* can be leveraged for cases involving any operator with a *Fixed* output type (i.e., the cases marked by either yellow or green) to further improve the performance of the memory access in the fused operator. More specifically, after fusing a sequence of layout transformation operators, these operators can be replaced by index computations for the operator it is fused with.

Strength reduction on index computation. To further reduce the overhead for index computation during data loading and storage, we propose the following optimizations for index computation. As a background, `Reshape` transforms an input with shape $[d_1, \ldots, d_m]$ into an output with shape $d'_1, \ldots, d'_n$ where the product of the new shape must equal to the old shape. `Transpose` permutes the dimensions according to a given order – formally: $Out_{i,\ldots,i_n} \leftarrow In_{\pi(i),\ldots,\pi(i_m)}$. However, it turns out that using the linear representation for all indexes directly leads to redundant computations, especially when multiple `Reshape` and `Transpose` operations are stacked together. Instead, our strength reduction strategy analyzes the index dependencies between consecutive layers. As shown in Figure 3, we define the index dependencies as *identity*, *split*, and *merge* based on static shape information in the computational graph. This allows us to transform indexes according to their dependency types using operands such as "//, %" (used in *split*) and "∗, +" (used in *merge*), "=" (used in *identity*). Since modular and divide operations are expensive on GPUs, we also simplify these terms by applying mathematical strength reduction rules. For example, if i, (C_a, C_b) are a variable index and constants respectively, then $i\%C_a\%C_b$ can be reduced to $i\%C_b$ when $C_a\%C_b \equiv 0$. This reduction commonly occurs when there are layout variations involved in index computation.

An additional point to be mentioned is that before index computation simplification, the fused operator is combining the logic of multiple layout transformations, rendering memory accesses fragmented and difficult to optimize. After the simplification process described earlier, the memory access pattern is more straightforward and exposes aggressive optimization opportunities.

3.2.2 A Reduction Dimension Based Layout Selection. At a high-level, we have the problem of selecting layout for all tensors throughout the computational graph. This global layout selection involves a large search space [45] and can be considered NP-hard, as evidenced by the connection to the Partitioned Boolean Quadratic Problem (PBQP) [2]. To keep the process at manageable costs, we develop a new heuristic solution based on *Reduction Dimension* that comprises two main steps. First, we conduct a local layout selection for tensors associated with individual edges in the computational graph (i.e., for pairs of operators), in which, the source operator is the producer while the sink is a consumer of a tensor. It is worth noting that we only need to handle the edges/operator pairs involving ILD&Variable (where we have denoted "search layout" in Tables 5 and 6). However, this process needs to be augmented through an additional step when we need to find the layout for producers with multiple consumers. Both steps rely on the knowledge of the *reduction dimension*, which we discuss next.

Our heuristic: reduction dimension. Reduction dimension(s) for an operand of an operator is the (set of) dimension along which data elements are involved in an aggregation. Take `MatMul` with $A_{i,k}$ and $B_{k,j}$ as inputs as an example. Its reduction dimension is k for both matrices A and B.

To examine how the notion of reduction dimensions is used, we revisit Tables 5 and 6. After multiple rounds of operator fusion and operator elimination, all preserved operators are ILD & Variable – this is because all operators in other types including ILI & Variable are fused into ILD & Variable eventually. The next step is to find the layout on each edge/-operator pair (with ILD & Variable type). Specifically, this step forces the first operator of each edge (i.e., producer) to generate the data layout preferred by the second operator (consumer). In our approach, this preferred data layout is decided by the reduction dimension of the second operator. For example, for the above `MatMul` example, we should keep the elements in both $A_{i,k}$ and $B_{k,j}$ along the reduction dimension (k) continuously stored in the memory. The insight here is that forcing the producer to generate a layout based on the reduction dimension of the consumer incurs relatively low added overhead as compared to other options. Specifically, sub-optimally writing results turns out to be better

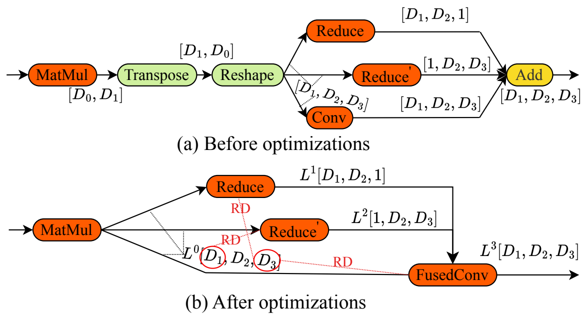


Figure 4. Examples of reduction dimensions and layout selection. D_0, D_1, D_2, D_3 represent dimensions in intermediate results. RD shorts for reduction dimension, and L means the memory layout. Add broadcasts its input shapes to match the shape of the largest one.

than sub-optimally reading input data. Microbenchmarking experiments comparing two versions (a. version that optimizes read performance, and b. version that optimizes write performance) using three operators (`Conv`, `MatMul`, and `Activation`) show speedups of 1.7×, 1.4×, and 1.1×, respectively for version a. `SmartMem` also includes a set of optimized tensor (matrix) layouts that are designed for both 1D memory and 2.5D memory for the producer to select. Section 3.3 elaborates these sample optimized layouts for 2.5D memory.

Global: layout selections based on reduction dimension. Searching for optimal layouts when one producer operator may have multiple consumer operators becomes challenging. We optimize the layout for the producer based on the collective needs from its consumers. Specifically, we optimize corresponding to the first k reduction dimensions required by the consumers, where k is the number of dimensions along which we can perform continuous memory access without any linearization and index computation. For example, $k = 2$ for 2.5D memory. Section 3.3 will elaborate more details for 2.5D memory. If consumers require more than k optimized layouts, `SmartMem` needs to maintain several copies of data with different layouts, and each layout is in this optimized combined format.

Example. Figure 4 illustrates our reduction dimension-based layout selection approach on a simplified computation graph. Figure 4(a) shows the original computation graph with a series of operators and the dimensions of their intermediate results. `Transpose` and `Reshape` (which splits the dimension D_0 into D_2 and D_3 for all successor operators) are inserted here for aligning the dimension for different kinds of operators (`MatMul` and `Conv`). Figure 4 (b) demonstrates an optimized computation graph with our reduction dimension-based layout selection. `Transpose` and `Reshape` are both ILD-Fixed and hence can be eliminated (as shown in Table 5). Then the output tensor of `MatMul` is consumed directly by `Reduce`, `Reduce'`, and `Conv(FusedConv)`; however, these operators suggest two reduction dimensions in which,

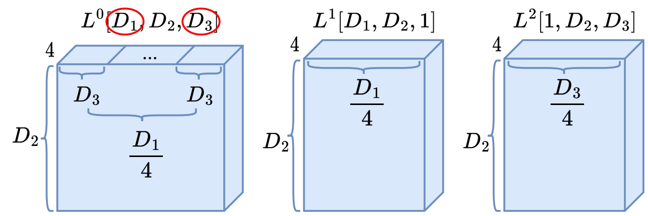


Figure 5. Sample layouts on 2.5D memory. Red circles denote reduction dimensions.

`Reduce'` has a reduction dimension of $D1$ while `Reduce` and `Conv(FusedConv)` have a common reduction dimension of $D3$. Assuming a mobile GPU with 2.5D memory (and cache), `SmartMem` can combine these two reduction dimension requirements (i.e., optimized layout requirements) for consumers of `MatMul` into an uniform data layout (L^0 as shown in the figure) and preferably map $D1$ and $D3$ to the two memory dimensions that allow continuous and direct indexed memory access. The layout generations of `Reduce` (L^1) and `Reduce'` (L^2) are more straightforward because they have no reduction dimension requirement. We will explain more details in Section 3.3.

3.3 Mapping Tensor to Texture Memory and Other Optimizations

2.5D memory (and corresponding dedicated read-only cache) on mobile GPUs allows more flexible index computation and eliminates index linearization if a tensor's dimension is less than 3. It also facilitates better exploring data reuse opportunities for 2D and 3D tensors. Thus, `SmartMem` also leverages 2.5D memory to further improve our optimized tensor layout design and memory access as mentioned earlier.

Optimized tensor layout on 2.5D memory. Figure 5 shows 3 sample tensor layouts when mapping a 3D tensor with varied shapes to 2.5D memory, corresponding to L_0, L_1, and L_2 in Figure 4, respectively (L_3 depends on its consumers). $D1$, $D2$, and $D3$ denote the dimensions of these tensors – both L^1 and L^2 have a dimension of size 1. Specifically, the red circles denote that these dimensions are specified as reduction dimensions by the consumer operators of this tensor, for example, L^0 has two reduction dimensions, $D1$ and $D3$ decided by `Reduce'` and `FusedConv` (in Figure 4), respectively, while L^1 and L^2 have no reduction dimensions.

It is beneficial to store the data along a reduction dimension continuously to improve data locality and allow better SIMD load and reduction operations. Thus, to map a tensor with two reduction dimensions (e.g., L^0 with $D1$ and $D3$) to 2.5D memory, `SmartMem` partitions one reduction dimension ($D1$). Each such partition has k elements ($k = 4$ in this example to match the size of the 0.5D in 2.5D) and stores $k\times$ <another reduction dimension> elements as a chunk along the dimensions of 2.5D memory that can be accessed in both directions. This layout results in efficient memory access and

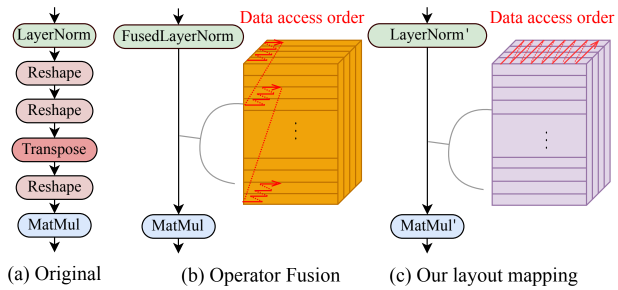


Figure 6. Data access patterns comparison before and after layout and access optimizations. In left (a), two consecutive Shape operations reshape the output from LayerNorm across different dimensions.

SIMD operations for consumer operators using either $D1$ or $D3$ as their reduction dimensions.

L^1 and L^2 do not have any reduction dimension because they are consumed by an element-wise addition operator (Add), so theoretically we are allowed to map either D1/D3 or D2 to that 0.5 dimension of 2.5D memory. However, because L^1 and L^2 are used together with L^0 in the FusedConv and their element-wise addition operations have been fused with the Conv, SmartMem maps them to 2.5D memory in a similar manner to L^0 (i.e., mapping D1 and D3 to the 0.5D of 2.5D memory, respectively), avoiding extra index computations.

Optimized memory access based on tensor layout. Figure 6 shows the high-level idea of our optimized data access on 2.5D memory by comparing the data access orders before and after this optimization. Figure 6 (a) shows the original computational graph, Figure 6 (b) shows the computational graph and data access order after the fusion and operator elimination offered by SmartMem, and Figure 6 (c) shows the computational graph and data access order after our optimized tensor layout mapping and memory access optimization. Before the optimized tensor layout mapping, although the Reshape and Transpose operations are fused (and eliminated), the memory access pattern is complex and fragmented, resulting in poor data locality (and similarly low SIMD efficiency). The optimized tensor layout mapping offers us an opportunity to access this tensor along its reduction dimension with a stride of 1, thus improving the data locality on 2.5D memory and cache, and parallel and SIMD efficiency on mobile GPUs.

Other optimizations. In addition to the previously mentioned optimizations, our framework incorporates an auto-tuning mechanism that utilizes Genetic Algorithms [51] for generating GPU execution configurations. These configurations include block dimensions, unrolling factors, and tiling shapes.

4 Evaluation

This section evaluates the SmartMem by comparing it to five state-of-the-art frameworks across different models. The objectives of this evaluation are as follows: 1) To exhibit the notable improvements SmartMem offers against existing, cutting-edge DNN frameworks on a mobile GPU, 2) To explore how various optimizations contribute to these performance enhancements, 3) To demonstrate the portability of proposed optimizations in SmartMem by evaluating the execution times on two other mobile platforms, and 4) To illustrate that our proposed optimizations enable performance improvement on a desktop-level GPU, which is less resource-constrained and has a traditional (one-dimensional) memory. Particularly, SmartMem is compared against MNN [32], NCNN [50], TFLite [1], TVM [10], and DNNFusion [51](refers as DNNF), on the mobile GPU.

4.1 Evaluation Setup

DNN Workloads: Our evaluation is conducted on 18 state-of-the-art DNN models with three different structures, including Transformer, ConvNet, and Hybrid (i.e., ones having both Transformer and ConvNet structures), as well as Stable Diffusion and LLMs. Table 7 characterizes them with a comparison of their type, attention mechanism, the number of parameters, the number of operators prior to optimizations, as well as the number of multiply-accumulate operations (MACs). We have 1) six Transformer models (AutoFormer [8, 9], CrossFormer [69], Swin [46], ViT [20], Stable Diffusion - TextEncoder [59], Pythia [6]), 2) four Convolution models (ConvNext [47], RegNet [58], ResNext [72], Yolo-V8 [33]), and 3) eight Hybrid models with both Transformer and Convolution structures (BiFormer [77], CSwin [19], EfficientVit [7], FlattenFormer [25], SMTFormer [43]), Conformer [24], StableDiffusion - UNet and VAEDecoder [59].

The Transformer structure can serve as a backbone for different CV and NLP tasks. Due to the space limitations, we report the results on the object detection task (for Yolo-V8) and image classification task (for all other models) in this evaluation. Since the training dataset has a negligible impact on the final inference latency, this section reports results from one training dataset for each model. Yolo-V8 is trained on MS COCO dataset [42]; Conformer, Stable Diffusion Models (SD-TextEncoder, SD-UNet, SD-VAEDecoder), and Pythia are from the pretrained checkpoints; other models are trained on ImageNet dataset [15]. Since the model accuracy is the same across all frameworks, our evaluation focuses only on execution latency.

Evaluation environment. SmartMem is built upon our previous work (DNNFusion [51]) that supports extensive operator fusion. We conduct our major experiments on a Oneplus cell phone using the high-end Qualcomm Snapdragon 8 Gen 2 platform [56], which includes a Qualcomm Kryo octa-core CPU and a Qualcomm Adreno 740 GPU with 16 GB of unified memory (shared by both CPU and GPU). In addition, to demonstrate our portability, we test SmartMem on an earlier generation of the Qualcomm platform - Snapdragon

Table 7. Model characterization and comparing the number of operators among frameworks. Hybrid means the combination of Transformer and ConvNet. "–" means the model is not supported by the framework yet.

Model	Model Type	Input Type	Attention	#Operators†	#Params (M)	#MACs (G)	#Operators with optimizations					
							MNN	NCNN	TFLite	TVM	DNNF	Ours
AutoFormer [8, 9]	Transformer	Image	Local	546	31.2	4.7	449	–	–	302	197	**148**
BiFormer [77]	Hybrid	Image	Local	2,042	25.5	4.5	1,189	–	–	1,029	602	**474**
CrossFormer [69]	Transformer	Image	Local	505	31	5	453	–	–	308	196	**155**
CSwin [19]	Hybrid	Image	Local	3,863	34.7	6.9	1,753	–	–	1,480	933	**604**
EfficientVit [7]	Hybrid	Image	Local	536	51	5.2	489	–	–	133	113	**101**
FlattenFormer [25]	Hybrid	Image	Local	2,016	37.3	7.2	1,558	–	–	918	665	**403**
SMTFormer [43]	Hybrid	Image	Local	1,406	22.5	4.9	1,905	–	–	844	469	**332**
Swin [46]	Transformer	Image	Local	765	28.9	4.6	596	–	–	374	207	**158**
ViT [20]	Transformer	Image	Global	444	102.8	21	379	–	–	289	168	**112**
Conformer [24]	Hybrid	Audio	Global	665	10	12	558	–	–	356	219	**163**
SD-TextEncoder [59]	Transformer	Text	Global	674	123	6.7	601	–	–	297	101	**84**
SD-UNet [59]	Hybrid	Image	Global	1,962	860	91	1355	–	–	889	436	**322**
SD-VAEDecoder [59]	Hybrid	Image	Global	287	50	312	206	–	–	156	103	**95**
Pythia [6]	Transformer	Text	Decoder	1,853	1,121	119	809	–	–	681	525	**355**
ConvNext [47]	ConvNet	Image	N/A	292	28.6	4.5	321	–	–	185	96	**81**
RegNet [58]	ConvNet	Image	N/A	282	19.4	3.2	197	282	197	155	122	**122**
ResNext [72]	ConvNet	Image	N/A	122	25	4.3	86	122	73	58	55	**55**
Yolo-V8 [33]	ConvNet	Image	N/A	233	3.2	4.4	176	233	–	88	75	**68**

#Operators† denotes the number of operators in the unoptimized computational graph.

835 [55], which has more limited resources, and on a MediaTek platform with Dimensity 700 SOCs. The Snapdragon 835 consists of an ARM Octa-core CPU, an Adreno 540 GPU, and 6 GB of unified memory. The MediaTek Dimensity 700 is equipped with an ARM Octa-core CPU, a Mali-G57 GPU, and 4 GB of unified memory. Furthermore, we evaluate our optimizations on an NVIDIA GPU to illustrate our generality. For this comparison, we implement our optimizations (excluding the mobile device-specific optimization for the 2.5D memory layout) in TorchInductor and compare it against the base version of TorchInductor. For all evaluated models and frameworks on mobile devices, GPU execution uses 16-bit floating-point representation. On desktop-level GPU, we use 32-bit floating-point representation for evaluation purposes. Noting that, we use 16-bit floating-point for mobile GPU because it is a common data type that all the frameworks support. Although other data types may have different accuracies, our optimizations are based on operator semantics thus not limited to specific data types. The batch size is set to 1 for all models unless otherwise specified. With the results we reported in this section, we have utilized the auto-tuning capabilities available in MNN, TVM, and TorchInductor to achieve the best possible performance. Each experiment is executed 50 times and only the average numbers are reported – as the variance was negligible, it is omitted for readability.

4.2 Overall Performance Comparison

Fusion rate comparison. Table 7 shows the model information with and without optimizations, including the number of operators in the unoptimized models as well as the number of operators in optimized versions produced by each framework ("–" indicates that the model is not supported by the framework). NCNN and TFLite do not support Transformer models on mobile GPU as they either lack support for key operators and/or do not reduce the memory requirements sufficiently. `SmartMem` achieves higher fusion rates compared to MNN and TVM for Transformer and Hybrid models, with ratios ranging from 2.2× to 7.2×, and 1.3× to 3.5× , respectively. For ConvNet models, `SmartMem` achieves fewer operators as compared to MNN, NCNN, TFLite, and TVM by a factor of 1.6× to 4.0×, 2.2× to 3.4×, 1.3× to 1.6×, and 1.1× to 2.3×, respectively. `SmartMem` offers greater benefits for Transformer and Hybrid models (e.g., BIformer, CSwin, SMTFormer, ViT) compared to ConvNet models (ResNext, RegNet, Yolo-V8). This is because Transformer/Hybrid models require more frequent data reshaping and transposing. The significant improvement in fusion rate stems from two separate components in `SmartMem`: the optimization based on DNNFusion which is achieving higher fusion rates compared to other frameworks, and the novel (reduction dimension-based elimination optimizations) introduced in this paper. Separating the two, we note that `SmartMem` achieves a fusion rate of up to 1.7× compared to DNNFusion. For three of the ConvNets (RegNet, ResNext, Yolo-V8), `SmartMem` demonstrates a similar fusion rate as DNNFusion as there are fewer Reshape and Transpose operations present. However, for the more complex ConvNet model ConvNext, `SmartMem` achieves an additional fusion opportunity of 1.2× compared to DNNFusion.

Table 8. End-to-end latency comparison across different frameworks – execution on GPU of Snapdragon 8 Gen 2. '–' means that the model is not supported. 'SD' represents for StableDiffusion, which includes three pipelines.

Model	#MACs (G)	Latency (ms) MNN	NCNN	TFLite	TVM	DNNF	**Ours**	Speed (GMACS) MNN	NCNN	TFLite	TVM	DNNF	**Ours**	**Speedup over DNNF**
AutoFormer	4.7	335	–	–	184	106	40.2	14	–	–	25	44	117	2.6×
BiFormer	4.5	1,736	–	–	208	186	56.1	3	–	–	22	24	81	3.3×
CrossFormer	5	336	–	–	156	121	38.2	15	–	–	32	41	130	3.2×
CSwin	6.9	703	–	–	261	225	57.6	10	–	–	26	31	120	3.9×
EfficientVit	5.2	208	–	–	243	112	22.5	25	–	–	21	47	232	5.0×
FlattenFormer	7.2	492	–	–	256	210	60.1	15	–	–	28	34	119	3.5×
SMTFormer	4.9	510	–	–	214	143	40	10	–	–	23	35	124	3.6×
Swin	4.6	372	–	–	158	135	30.6	15	–	–	29	34	149	4.4×
ViT	21	533	–	–	1,050	277	103	39	–	–	20	76	204	2.7×
Conformer	12	1,736	–	–	863	284	106	7	–	–	14	42	113	2.7×
SD-TextEncoder	6.7	153	–	–	216	73	38	44	–	–	31	92	176	1.9×
SD-UNet	90	2,172	–	–	3,969	1,108	412	42	–	–	23	82	219	2.7×
SD-VAEDecoder	312	2,730	–	–	5,663	1,596	866	114	–	–	55	195	360	1.8×
Pythia	119	3,034	–	–	6,602	1,489	663	40	–	–	18	80	180	2.3×
ConvNext	4.5	271	–	–	5,543	109	33.4	17	–	–	0.81	41	135	3.3×
RegNet	3.2	61	33	36.4	71	31	24.7	52	106	88	45	103	129	1.3×
ResNext	4.3	158	38	66	106	33	15.7	27	111	64	40	129	271	2.1×
Yolo-V8	4.4	32	28	–	141	26	22	138	169	–	31	169	200	1.2×
Geo-mean (speedup)		7.9×	1.6×	2.5×	6.9×	2.8×	1.0×	7.9×	1.6×	2.5×	6.9×	2.8×	1.0×	-

The results collected from MNN/TVM/DNNFusion with auto-tuning.

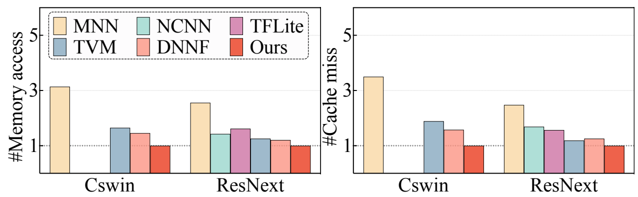
Furthermore, for Transformer and Hybrid models,`SmartMem` enables 1.1× to 1.7× more fusion opportunities through our elimination techniques when compared to DNNFusion.

End-to-end execution time comparison. Table 8 presents a comparison of overall execution latency and speed (measured in GMACS) among six frameworks on our target mobile GPU. In the case of Transformer and Hybrid models, `SmartMem` achieves 3.2× to 30.9×, and 3.7× to 10.8× speedups compared with MNN and TVM, respectively (NCNN and TFLite do not support Transformer and Hybrid models on mobile GPU). As for ConvNet models, `SmartMem` achieves 1.5× to 10.1×, 1.3× to 2.4×, 1.5× to 4.2×, and 2.9× to 166× compared with MNN, NCNN, TFLite, and TVM, respectively. The exceptionally large speedup on ConvNext when compared to TVM is due to TVM lacking an efficient layout design for a reduction operator `GroupConvolution`. For BIFormer, `SmartMem` significantly outperforms MNN because the token selection mechanism in BIFormer leads to more data transformation operations, resulting in more benefits from eliminating layout transformations. Compared to DNNFusion, `SmartMem` achieves 1.8× to 5.0× speed improvement for Transformer and Hybrid models, while achieving 1.2× to 3.3× improvement for ConvNets. The gains on ConvNets compared with DNNFusion are because our layout selection enables more hardware-friendly data access and computation pattern across multiple operators. It is worth noting that `SmartMem` delivers similar speed (around 120 GMACS) for models with similar structures regardless of the number of operators, i.e., for AutoFormer, BIFormer, CrossFormer, CSwin, FlattenFormer, SMTFormer, and Swin. On the other hand, for EfficientViT, ViT, ResNext, and Yolo-V8 models, the speed is higher (over 200 GMACS). This is because these models either have more intensive computation in each operator (e.g., ViT and EfficientViT), which means more data reuse in computation or have a more regular computation pattern (RegNet and ResNext) with pure Convolution structure.

Memory performance (overall). Figure 7(a) compares the number of memory accesses (left) and the number of cache misses (right). The results are normalized by `Ours` (`SmartMem`) for readability. Due to space limitations, we only present results for one local attention Hybrid model (Cswin) and one ConvNet model (ResNext). Cswin execution using our framework is compared against MNN, TVM, and DNNFusion, whereas ResNext execution is compared against these as well as NCNN and TFLite. As shown in Figure 7, `SmartMem` utilizes 1.8× fewer memory accesses on average and achieves an average of 2.0× fewer cache misses compared to other frameworks. In the following section, we further study the impact of our key optimizations on memory accesses.

4.3 Optimization Breakdown and Analysis

This section studies the impact of the key optimizations in `SmartMem` on execution latency. We evaluate the gains from each optimization incrementally over our baseline version



(a) Memory access count **(b)** Cache miss count

Figure 7. Memory access count and cache miss count comparison compared with other frameworks. The memory access data is collected from the hardware counter on the mobile GPU, showing the total number of data accesses happening in the global memory of the mobile GPU. All results are normalized by ours.

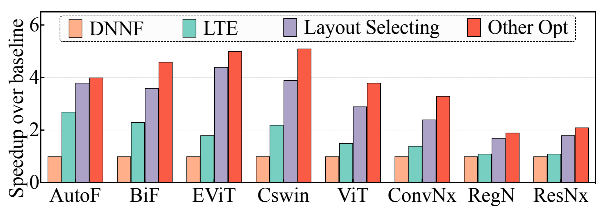
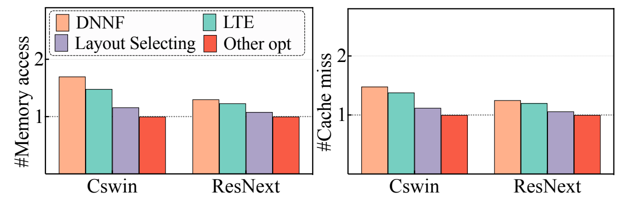


Figure 8. Performance breakdown analysis: speedup over baseline (DNNFusion).

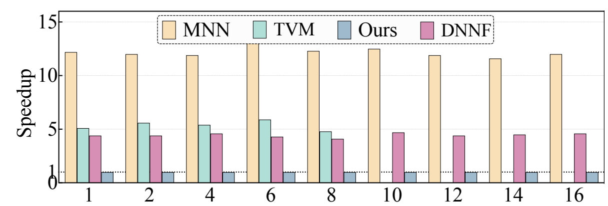
DNNF (shorts for DNNFusion). Figure 8 illustrates the impact of our proposed optimizations on latency with eight models. These include five Transformer and Hybrid models (AutoFormer (AutoF), BiFormer (BiF), EfficientViT (EViT), Cswin, ViT) and three ConvNet models (ConvNext(ConvNx), RegNet (RegN), ResNext (ResNx)) on the mobile GPU. LTE shorts for Layout Transformation Elimination. Due to space limitations, we omit experiments on other models that show a similar trend. For Transformer and Hybrid, compared to DNNF, Layout Transformation Elimination (LTE) achieves a speedup of 1.5× to 2.7×. Layout Selecting brings an additional speedup of 1.4× to 1.9×, and other optimizations (texture memory-related and tuning) bring an additional speedup of 1.2× to 1.4×. For ConvNet, these numbers are 1.1× to 1.4×, 1.5× to 1.7×, and 1.1× to 1.4×, respectively. Considering Index Comprehension, for Transformer and Hybrid models, it contributes to the speedup results of Layout Transformation Elimination by 1.1× to 1.3×, while for ConvNets, it contributes between 1.1× and 1.2×.

Memory and cache breakdown. To further investigate the impact of our key optimizations on the underlying hardware, Figure 9 illustrates the breakdown of memory and cache usage in `SmartMem`. Figure 9 demonstrates the reduction in memory access counts (left) and cache miss counts (right) for Cswin and ResNext, respectively. Layout Transformation Elimination has a greater effect on reducing memory access count than cache miss count because it eliminates the need



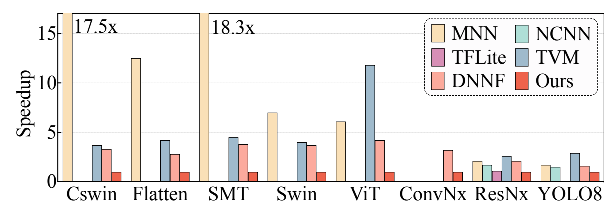
(a) Memory access counts **(b)** Cache miss counts

Figure 9. Optimization breakdown: memory access count and cache miss count comparison.

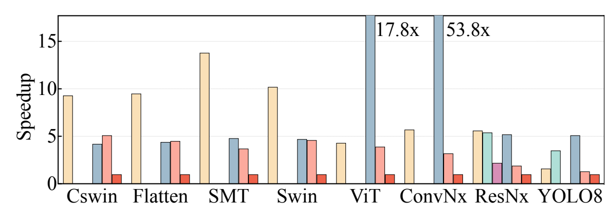


(a) Swin

Figure 10. Performance comparison with different batch sizes for Swin. We report the speedup for better readability.



(a) On Mali-G57 (Mediatek Dimensity 700)



(b) On Adreno 540 (Qualcomm Snapdragon 835)

Figure 11. Portability evaluation on Mediatek Dimensity 700 and Qualcomm Snapdragon 835. We report the speedup for better readability. Empty bar on other frameworks means the model is not supported due to lack of operator supports or insufficient memory on device.

for data reorganization and enables more fusion opportunities. On the other hand, Layout Selection has a greater impact on cache miss count than memory access count as it designs better layouts for each operator thus enabling better data access patterns.

Table 9. Comparison of end-to-end execution latency on NVIDIA GPU between PytorchInductor and `SmartMem` **for Swin and SMTFormer.** Results are collected with a batch size of 1.

Model	System	Device	TorchInductor (ms)	Ours (ms)
Swin	Ubuntu 20.04	Tesla V100	7.5	6.1
AutoFormer	Ubuntu 20.04	Tesla V100	5.1	4.6

4.4 Performance Impact of Batch Size

Figure 10 illustrates the performance improvements achieved by `SmartMem` on mobile GPU, compared to MNN, TVM, and DNNF, across different batch sizes. We report the results for a representative Transformer model, Swin with batch sizes ranging from 1 to 16. Other models show a similar trend. An empty bar for other frameworks indicates that the corresponding batch size is not supported due to insufficient device memory. For different batch sizes, `SmartMem` shows speed improvements of 11.6× to 13.2×, 4.8× to 5.9×, and 4.1× to 4.7× compared to MNN, TVM, and DNNFusion, demonstrating its scalability.

4.5 Portability Evaluation

Older Mobile Platforms: Figure 11 shows the speedup achieved on Mediatek Dimensity 700 and Qualcomm Snapdragon 835. MNN and TVM do not support ConvNext on Mali-G57 due to insufficient memory (4GB unified memory). `SmartMem` achieves similar speedup on these platforms with limited resources, and is actually less sensitive to having fewer resources because our Layout Transformation Elimination and Layout Selection reduces the overall number of operators, resulting in lower memory and cache pressure.

Desktop-level GPU: We have implemented our Layout Transformation Elimination and Layout Selecting on TorchInductor and conducted a performance comparison. TorchInductor automatically selects the optimal implementation from either TensorRT or a Pytorch built-in or compiler-generated Triton kernel. We compared the performance of two Transformer models, Swin and AutoFormer, in this environment. As shown in Table 9, `SmartMem`, achieves 1.23× and 1.11× better execution efficiency than TorchInductor with data type FP32. As these results are from a quick and separate implementation, we expect that additional benefits will be possible in future work.

4.6 Discussion

Limitations. While our optimizations significantly improve the performance of transformer models with many layout transformation operations, it is worth pointing out that `SmartMem` yields only modest performance improvements for traditional convolutional neural networks such as Yolo-V8 and RegNet, or models in general with few layout transformation operations compared to DNNF. In comparison

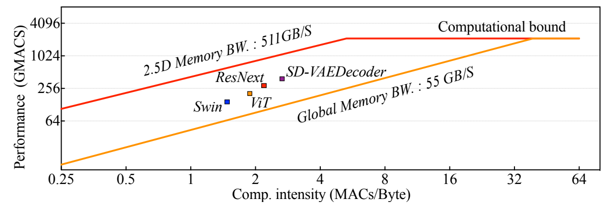


Figure 12. Roofline analysis for peak and actual performance (GMACs/s) on mobile GPU for ViT.

to DNNF (the best baseline shown in Table 8) in CNN-only models, particularly RegNet and Yolo-V8, `SmartMem` achieves a speedup of 1.3× and 1.2×, respectively. These two models primarily consist of computationally intensive operators (Conv2D), and the advanced operator fusion proposed in DNNF works effectively. Conversely, for CNNs with more layout transformation operations (ConvNext) or those sensitive to the data layouts like ResNext, `SmartMem` delivers a speedup of 3.3× and 2.1×, demonstrating the effectiveness of our optimization in eliminating layout transformations and selection for these models.

Roofline analysis for the achieved performance. To better evaluate `SmartMem`'s optimization effect, Figure 12 shows the results from roofline analysis [70] on four models (Swin, ViT, ResNext, and SD-VAEDecoder). Here, the yellow line and red line show the theoretical peak performance assuming that all data is accessed from global memory and 2.5D texture memory, respectively. The global memory and texture memory bandwidths are 55 GB/s and 511 GB/s, respectively, and the mobile GPU's peak computation performance is 2.0 TMACs/s [41]. These models consist of various operators with different computation intensity levels. Certain operators have a low computation intensity (e.g., activation functions and elementwise arithmetic operations), while others exhibit a high computation intensity (e.g., `Conv` and `MatMul`). For ease of comparison, the x-axis in this analysis denotes the computation intensity (MACs/byte) averaged across the entire computational graph.

Figure 12 shows the analysis for Swin, ViT, ResNext, and SD-VAEDecoder, which all have different levels of computational intensity, achieve 149, 204, 271, and 360 GMACs/s, respectively. Specifically, Swin performs the worst while SD-VAEDecoder performs the best because Swin contains more low computation intensity operators whereas SD-VAEDecoder contains more high computation intensity operators. Under an unrealistic assumption that all data accesses are from 2.5D texture memory, `SmartMem` achieves 24%, 27%, 31%, and 35% of the theoretical peak performance. A higher computation intensity of a model implies more `Conv` and/or `MatMul` operators in the model. These operators are more likely to read data from 2.5D texture memory, thus achieving a higher percentage of the theoretical peak performance.

Impact of redundant data copies with different layouts. The SmartMem stores redundant copies of data, specifically intermediate results, with different layouts when an operator has multiple consumers requiring varied input data layouts. During runtime, these redundant layouts do not significantly increase memory consumption for the models we evaluated. This is primarily due to two reasons. Firstly, maintaining multiple copies of data in different layouts is not commonly seen during execution; for instance, Swin and ViT have a maximum active redundant copy size of 3.0MB and 2.3MB, respectively. Additionally, similar to TVM, our implementation allocates intermediate results from a memory pool allowing efficient reuse of memory resources by releasing data copies back into the pool when they are no longer needed by any consumers. In addition, compared to DNNFusion, eliminating kernels reduces further memory usage. Take SwinTransformer and ViT as examples: SmartMem decreases the number of operators by 24% and 33% compared to DNNF, as shows in Table 7, resulting in respective reductions in memory consumption of 14% and 15%.

5 Related Work

Operator fusion and layout optimizations. Operator fusion is a commonly used optimization technique in many advanced DNN inference frameworks. In the past, MNN [32], TFLite [1], NCNN [50], and Pytorch-Mobile [53] have all employed fixed-pattern fusion, which is based on identifying specific operator combinations to be fused (and thus limiting fusion for those cases). TVM [10] recently started supporting operator fusion with more general rules by classifying the operator into three categories [66]: 1) Layout agnostic (e.g., ReLU), 2) Lightly-layout sensitive (e.g., Reduce), and 3) Heavily-layout sensitive (e.g., Conv). DNNFusion [51] generates fusion plans by classifying operators and their combinations, allowing for a wider range of fusion optimizations. However, it cannot eliminate explicit data transformation operators through improved layouts. Furthermore, with our Layout Transformation Elimination, SmartMem offers even more opportunities for operator fusion compared to DNNFusion. Our evaluation demonstrates superior results in terms of fusion rate and latency. Ivanov et al. [29] aim to reduce data movement in Transformer training on desktop GPUs, and similar to TVM, they propose an optimization framework that systematically fuses operators and selects data layouts by classifying the operator into three categories. Different types of operators can only be fused if they have the exact same iteration space, except for the reduction dimension, including both the order and size of the dimensions. This approach cannot optimize (including eliminating layout transformations) cases involving frequent changes in dimension order and size, which Reshape and Transpose kind of operations explicitly do.

DNN inference frameworks on mobile. As AI applications on mobile devices continue to grow, there is a strong emphasis on optimizing DNN inference frameworks for mobile platforms. Efforts such as DeepEar [39], DeepSense [75], Mobisr [40], and others [26, 28, 38, 64, 73], have primarily focused on specific tasks or traditional convolutional neural networks, or require special hardware support. There are other research directions that target DNN scheduling for heterogeneous platforms, including Band [30], BlastNet [44], CoDL [31], and others [63, 71].

Other DNN execution optimization frameworks. The advancement of Artificial General Intelligence (AGI) has been significantly bolstered by the evolution of Transformer-based models, paralleled by notable progress in supporting acceleration frameworks like FlashAttention [13, 14], and vLLM [37]. FlashAttention improves the attention mechanism in Transformers by using tiling strategies to minimize memory access between the GPU's high bandwidth memory (HBM) and on-chip SRAM. However, FlashAttention requires extensive register usage in its kernel, which limits its effectiveness on mobile GPUs with limited register capacity [11]. SmartMem focuses on Layout Transformation Elimination, essentially complementing their work – combining two approaches could be an interesting future direction. vLLM is an efficient LLMs serving system targeting memory fragmentation issues, inspired by the classical virtual memory and paging techniques used in operating systems. vLLMs proposes PagedAttention to enhance processing throughput and capability for long sequences in LLMs, achieving near-zero waste in KV cache memory. TensorFlow-XLA [23] and TorchInductor [54] are advanced compilers that translate computational graphs and linear algebra expressions into machine code. TensorFlow-XLA uses a "pad-reshape-transpose" strategy to optimize layout transformation and tiling, which can lead to additional overhead for layout conversion. On the other hand, TorchInductor relies on pre-assigned layouts of specific operators or satisfies layout constraints from library calls (e.g., TensorRT).

6 Discussion and Future Work

Automating operator categorization. The classification of operators within SmartMem is designed to evaluate their sensitivity to variations in both computational pattern and data access pattern (i.e., computation performance and output layout). By utilizing established intermediate representations like tensor expressions supported by TVM, it is possible to create a tool that automates and adapts for operator categorization. This approach could be further enhanced through the integration of symbolic manipulation tools, enabling the handling of more intricate scenarios. Our future work will further enhance this process and explore the potential of automating the categorization of operators.

Layout optimizations beyond mobile devices. SmartMem optimizes data layout transformations commonly seen in modern DNN computational graphs, leading to significant performance improvements across various DNNs, including LLMs. The optimizations in SmartMem, such as Layout Transformation Elimination and layout selection, are also applicable across different platforms, including desktop-level GPUs. However, mobile GPUs benefit more from SmartMem. This is because mobile GPUs have more restricted memory size and bandwidth, which makes their performance more sensitive to data transformation elimination optimizations. In addition, mobile GPUs rely on an efficient 2.5D texture memory for data processing, which offers us further data layout optimization opportunities. Implementations on desktop-level GPUs mainly rely on shared memory and cache, and their texture memory usage is relatively restricted. SmartMem also has the potential to be extended to other platforms, such as FPGAs and ASICs, where the memory hierarchy and data layout optimizations are also critical for performance.

7 Conclusions

This paper introduces SmartMem, a novel framework to significantly improve the performance of DNN execution on mobile devices for both CNN and Transformer structures. SmartMem categorizes DNN operators into four groups based on their input/output layouts and computations, considers combinations of producer-consumer edges between the operators, and searches optimized layouts with multiple carefully designed methods. Our extensive experiments with 18 cutting-edge models show significant speedup compared to MNN, NCNN, TFLite, TVM, and DNNFusion. Looking ahead, this work paves the way for further research in optimizing LLMs (with larger parameters and more operators) on mobile devices. Considering the nature of mobile GPUs (e.g., limited memory bandwidth and register file size), future work of SmartMem will explore combining different optimizations (e.g., pruning, quantization) that dynamically adapt to the ever-changing landscape of AGI and mobile hardware.

Acknowledgments

The authors want to extend their appreciation to the anonymous reviewers and shepherd, Tim Harris, for their valuable and thorough feedback. All of these constructive suggestions have greatly contributed to enhancing this paper. This work was supported in part by the National Science Foundation (NSF) under the awards of CCF-2047516 (CAREER), CCF-2146873, CCF-2333895, CCF-2334273, CNS-2312207, CNS-2230944, CNS-2341378, III-2008557, OAC-2333899, and DOE DE-SC0021293. Any errors and opinions are not those of the NSF and are attributable solely to the author(s). The authors also acknowledge William & Mary Research Computing for providing computational resources.

References

[1] Martin 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 G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI 2016*, pages 265–283, USA, 2016. USENIX Association.

[2] Andrew Anderson and David Gregg. Optimal dnn primitive selection with partitioned boolean quadratic programming. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 340–351, 2018.

[3] Anurag Arnab, Mostafa Dehghani, Georg Heigold, Chen Sun, Mario Lučić, and Cordelia Schmid. Vivit: A video vision transformer. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6836–6846, 2021.

[4] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[5] Gedas Bertasius, Heng Wang, and Lorenzo Torresani. Is space-time attention all you need for video understanding? In *ICML*, volume 2, page 4, 2021.

[6] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pages 2397–2430. PMLR, 2023.

[7] Han Cai, Chuang Gan, and Song Han. Efficientvit: Enhanced linear attention for high-resolution low-computation visual recognition. *arXiv preprint arXiv:2205.14756*, 2022.

[8] Minghao Chen, Houwen Peng, Jianlong Fu, and Haibin Ling. Autoformer: Searching transformers for visual recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 12270–12280, October 2021.

[9] Minghao Chen, Kan Wu, Bolin Ni, Houwen Peng, Bei Liu, Jianlong Fu, Hongyang Chao, and Haibin Ling. Searching the search space of vision transformer. *Advances in Neural Information Processing Systems*, 34:8714–8726, 2021.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *OSDI 2018*, pages 578–594, 2018.

[11] Yu-Hui Chen, Raman Sarokin, Juhyun Lee, Jiuqiang Tang, Chuo-Ling Chang, Andrei Kulik, and Matthias Grundmann. Speed is all you need: On-device acceleration of large diffusion models via gpu-aware optimizations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4650–4654, 2023.

[12] Zihang Dai, Hanxiao Liu, Quoc V Le, and Mingxing Tan. Coatnet: Marrying convolution and attention for all data sizes. *Advances in neural information processing systems*, 34:3965–3977, 2021.

[13] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[14] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*, pages 248–255. IEEE Computer Society, 2009.

[16] Erik Derner and Kristina Batistič. Beyond the safeguards: Exploring the security risks of chatgpt, 2023.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, 2019. Association for Computational Linguistics.

[18] Lei Ding, Hao Tang, and Lorenzo Bruzzone. Lanet: Local attention embedding to improve the semantic segmentation of remote sensing images. *IEEE Transactions on Geoscience and Remote Sensing*, 59(1):426–435, 2020.

[19] Xiaoyi Dong, Jianmin Bao, Dongdong Chen, Weiming Zhang, Nenghai Yu, Lu Yuan, Dong Chen, and Baining Guo. Cswin transformer: A general vision transformer backbone with cross-shaped windows, 2021.

[20] 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. An image is worth 16x16 words: Transformers for image recognition at scale, 2020.

[21] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. Glm: General language model pretraining with autoregressive blank infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 320–335, 2022.

[22] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 804–817, 2023.

[23] Google. Tensorflow xla. https://www.tensorflow.org/xla, 2023.

[24] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, and Ruoming Pang. Conformer: Convolution-augmented transformer for speech recognition. In Helen Meng, Bo Xu, and Thomas Fang Zheng, editors, *Interspeech 2020, 21st Annual Conference of the International Speech Communication Association, Virtual Event, Shanghai, China, 25-29 October 2020*, pages 5036–5040. ISCA, 2020.

[25] Dongchen Han, Xuran Pan, Yizeng Han, Shiji Song, and Gao Huang. Flatten transformer: Vision transformer using focused linear attention. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023.

[26] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 123–136. ACM, 2016.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 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*, pages 770–778. IEEE Computer Society, 2016.

[28] Loc N Huynh, Youngki Lee, and Rajesh Krishna Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 82–95. ACM, 2017.

[29] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021.

[30] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. Band: coordinated multi-dnn inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 235–247, 2022.

[31] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. Codl: efficient cpu-gpu co-execution for deep learning inference on mobile devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 209–221. Association for Computing Machinery New York, NY, USA, 2022.

[32] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, et al. Mnn: A universal and efficient inference engine. *Proceedings of Machine Learning and Systems*, 2:1–13, 2020.

[33] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. Ultralytics yolov8, 2023.

[34] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European conference on computer vision*, pages 694–711. Springer, 2016.

[35] Mahmut Kandemir, Alok Choudhary, Jagannathan Ramanujam, and Prithviraj Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 285–296. IEEE, 1998.

[36] Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W Mahoney, et al. Full stack optimization of transformer inference. In *Architecture and System Support for Transformer Models (ASSYST@ ISCA 2023)*, 2023.

[37] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[38] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–12, 2016.

[39] Nicholas D Lane, Petko Georgiev, and Lorena Qendro. Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM international joint conference on pervasive and ubiquitous computing*, pages 283–294, 2015.

[40] Royson Lee, Stylianos I Venieris, Lukasz Dudziak, Sourav Bhattacharya, and Nicholas D Lane. Mobisr: Efficient on-device super-resolution through heterogeneous mobile processors. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[41] Rendong Liang, Ting Cao, Jicheng Wen, Manni Wang, Yang Wang, Jianhua Zou, and Yunxin Liu. Romou: Rapidly generate high-performance tensor kernels for mobile gpus. In *The 28th Annual International Conference On Mobile Computing And Networking (MobiCom 2022)*. ACM, February 2022.

[42] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Doll'a r, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[43] Weifeng Lin, Ziheng Wu, Jiayu Chen, Jun Huang, and Lianwen Jin. Scale-aware modulation meet transformer, 2023.

[44] Neiwen Ling, Xuan Huang, Zhihe Zhao, Nan Guan, Zhenyu Yan, and Guoliang Xing. Blastnet: Exploiting duo-blocks for cross-processor real-time dnn inference. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, pages 91–105, 2022.

[45] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing {CNN} model inference on {CPUs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1025–1040, 2019.

[46] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[47] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[48] Lydia Manikonda, Aditya Deotale, and Subbarao Kambhampati. What's up with privacy? user preferences and privacy concerns in intelligent personal assistants. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, pages 229–235, 2018.

[49] Naraig Manjikian and Tarek S Abdelrahman. Fusion of loops for parallelism and locality. *IEEE transactions on parallel and distributed systems*, 8(2):193–209, 1997.

[50] Hui Ni and The ncnn contributors. ncnn, June 2017.

[51] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021.

[52] Michael FP O'Boyle and Peter MW Knijnenburg. Integrating loop and data transformations for global optimization. *Journal of Parallel and Distributed Computing*, 62(4):563–590, 2002.

[53] 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. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[54] PyTorch. TorchInductor, January 2022.

[55] Qualcomm. Snapdragon 835. https://www.qualcomm.com/products/snapdragon-835-mobile-platform, 2016.

[56] Qualcomm. Snapdragon 845. https://en.wikipedia.org/wiki/List_of_Qualcomm_Snapdragon_systems_on_chips, 2017.

[57] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[58] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. Designing network design spaces. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10428–10436, 2020.

[59] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10684–10695, 2022.

[60] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. Nimble: Efficiently compiling dynamic neural networks for model inference. *Proceedings of Machine Learning and Systems*, 3:208–222, 2021.

[61] Jun Shirako and Vivek Sarkar. An affine scheduling framework for integrating data layout and loop transformations. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 3–19. Springer, 2020.

[62] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. Pure tensor program rewriting via access patterns (representation pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, pages 21–31, 2021.

[63] Hsin-Hsuan Sung, Jou-An Chen, Wei Niu, Jiexiong Guan, Bin Ren, and Xipeng Shen. Decentralized {Application-Level} adaptive scheduling for {Multi-Instance}{DNNs} on open mobile devices. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 865–877, 2023.

[64] Tianxiang Tan and Guohong Cao. Efficient execution of deep neural networks on mobile devices with npu. In *Proceedings of the 20th International Conference on Information Processing in Sensor Networks (Co-Located with CPS-IoT Week 2021)*, pages 283–298, 2021.

[65] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[66] Apache TVM. Convertlayout pass, 2020.

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

[68] Huiyu Wang, Yukun Zhu, Bradley Green, Hartwig Adam, Alan Yuille, and Liang-Chieh Chen. Axial-deeplab: Stand-alone axial-attention for panoptic segmentation. In *European conference on computer vision*, pages 108–126. Springer, 2020.

[69] Wenxiao Wang, Lu Yao, Long Chen, Binbin Lin, Deng Cai, Xiaofei He, and Wei Liu. Crossformer: A versatile vision transformer hinging on cross-scale attention. In *International Conference on Learning Representations, ICLR*, 2022.

[70] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[71] Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 392–405, 2019.

[72] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

[73] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. Deepcache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, page 129–144, New York, NY, USA, 2018. Association for Computing Machinery.

[74] Weijian Xu, Yifan Xu, Tyler Chang, and Zhuowen Tu. Co-scale conv-attentional image transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9981–9990, 2021.

[75] Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, and Tarek F. Abdelzaher. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In Rick Barrett, Rick Cummings, Eugene Agichtein, and Evgeniy Gabrilovich, editors, *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, pages 351–360. ACM, 2017.

[76] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[77] Lei Zhu, Xinjiang Wang, Zhanghan Ke, Wayne Zhang, and Rynson Lau. Biformer: Vision transformer with bi-level routing attention. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.