Stitching Weight-Shared Deep Neural Networks for Efficient Multitask Inference on GPU

Zeyu Wang*, Xiaoxi He†, Zimu Zhou‡, Xu Wang*, Qiang Ma*,
Xin Miao*, Zhuo Liu§, Lothar Thiele† and Zheng Yang*
*School of Software and BNRist, Tsinghua University
† Computer Engineering and Networks Laboratory, ETH Zurich
‡School of Computing and Information Systems, Singapore Management University
§Remarkable Universal Network Company
wzy20@mails.tsinghua.edu.cn, {hex, thiele}@ethz.ch, zimuzhou@smu.edu.sg
{wangxu2020, yangzheng, miaoxin}@tsinghua.edu.cn, {tsinghuamq, zhuoliumail}@gmail.com

***Abstract*—Intelligent personal and home applications demand multiple deep neural networks (DNNs) running on resource-constrained platforms for compound inference tasks, known as multitask inference. To fit multiple DNNs into low-resource devices, emerging techniques resort to weight sharing among DNNs to reduce their storage. However, such reduction in storage fails to translate into efficient execution on common accelerators such as GPUs. Most DNN graph rewriters are blind for multi-DNN optimization, while GPU vendors provide inefficient APIs for parallel multi-DNN execution at runtime. A few prior graph rewriters suggest cross-model graph fusion for low-latency multi-DNN execution. Yet they request duplication of the shared weights, erasing the memory saving of weight-shared DNNs. In this paper, we propose `MTS`, a novel graph rewriter for efficient multitask inference with weight-shared DNNs. `MTS` adopts a model stitching algorithm which outputs a single computational graph for weight-shared DNNs without duplicating any shared weight. `MTS` also utilizes a model grouping strategy to avoid overwhelming the GPU when co-running tens of DNNs. Extensive experiments show that `MTS` accelerates multitask inference by up to 6.0× compared to sequentially executing multiple weight-shared DNNs. `MTS` also yields up to 2.5× lower latency and 3.7× less memory usage compared with `NETFUSE`, a state-of-the-art multi-DNN graph rewriter.**

***Index Terms*—Deep Neural Networks; Multitask Inference; Model Acceleration**

I. Introduction

Deep learning empowered ubiquitous applications increasingly demand the co-execution of *multiple* deep neural networks (DNNs), known as *multitask inference*, for complex cognitive analysis [1]–[4]. In multitask inference, multiple DNNs, each pre-trained for a single inference task, run concurrently on resource-constrained platforms ranging from edge servers [5] to embedded devices [1]–[3], [6] for *correlated* inference tasks. Such multitask inference is critical for future applications such as smart glasses that identify user attributes *e.g.*, age, gender, face, and recognize objects [2], [6]–[8], personal robots that classify places and sounds [3], autonomous vehicles that perceive the surroundings with front, side, rear camera views [4], home hubs that recognize emotions from speech and facial expression [9] etc.

For efficient execution on low-resource platforms, DNNs often undergo multiple levels of optimizations. At the *model* level, over-parameterised DNNs can be compressed without loss in inference accuracy [10]. The compressed DNNs, typically represented as computational directed acyclic graphs (DAGs), are then optimized at the *graph* level via sub-graph fusion and substitution to generate functionally equivalent yet faster DAGs for the target hardware platform [11], [12]. The DAGs can be further optimized at the *runtime* level for better resource utilization via hardware-aware scheduling [13]. Mainstream deep learning development frameworks such as TensorFlow and PyTorch support automatic and customized model- or graph-level optimizations whereas hardware vendors like NVIDIA also provide APIs for user-specified runtime-level accelerations. Despite extensive research on efficient DNN execution [14]–[17], most efforts only focus on accelerations *within a single model*, overlooking the potential gains from *cross-model* optimization.

An emerging technique for efficient multitask inference is cross-model weight sharing [3], [6], [8], [9], [18], [19]. Sharing weights across DNNs pre-trained for correlated tasks reduces the memory footprint to deploy them on low-memory devices. Task correlation is pervasive since multiple DNNs may take the same input to generate different labels, or augment complimentary inputs to jointly output a single label. For example, DNNs that identify user age and faces from the same input image may extract similar low-level features, while DNNs for video- and audio-based emotion recognition may share similar high-level features. As illustrated in Fig. 1(b), cross-model weight-sharing methods automatically identify correlated weights (colored in green) among weight matrices pre-trained for different tasks (see Fig. 1(a)). Such weight-shared DNNs save the storage for multitask inference.

However, the memory saving of weight-shared DNNs fails to translate into efficient execution with existing graph- and runtime-level optimizations. On the one hand, popular graph rewriters such as TVM [11] and NVIDIA TensorRT [12] optimize each DAG in isolation (see Fig. 1(c)). Such graph rewriting duplicates the shared weights to create independent DAGs for each task. On the other hand, native runtime APIs such as CUDA Stream [20] and NVIDIA MPS [21] offer limited multi-DNN parallelism support. Executing individual

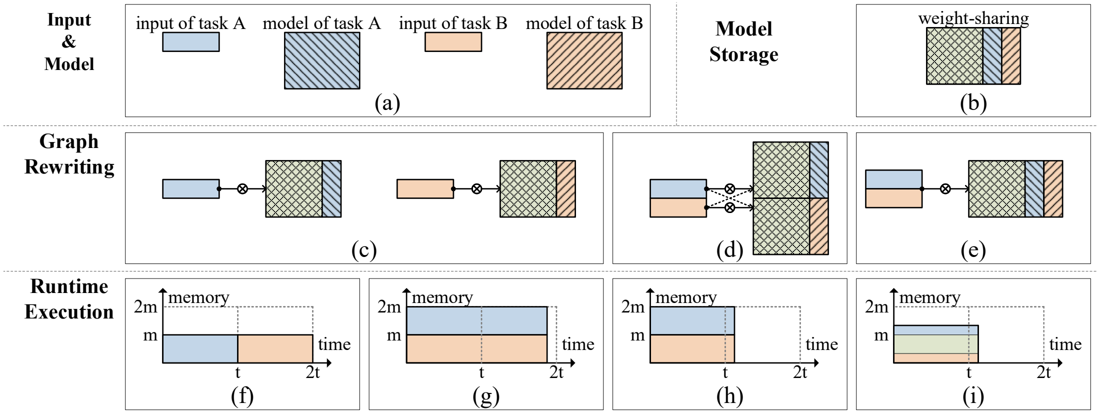


Fig. 1. Executing weight-shared neural networks on a GPU. (a) Input and pre-trained single-task models for two correlated tasks 1 and 2. (b) Cross-model weight-sharing for storage saving of models. The green portion represents the shared weights uncovered by techniques such as [3], [6], [8]. Multitask inference on weight-shared networks can be compiled as (c) two separate computational graphs; (d) a single computational graph by duplicating shared weights; or (e) a single computational graph without duplicating shared weights. At runtime, (f) sequential execution of two graphs incurs long latency; (g) parallel execution of two graphs leads to both high memory footprint and long latency due to inefficient GPU runtime support; (h) the state-of-the-art executes a combined graph with a single stream, but duplicates the shared weights. (i) Our objective is to achieve both low memory and low latency.

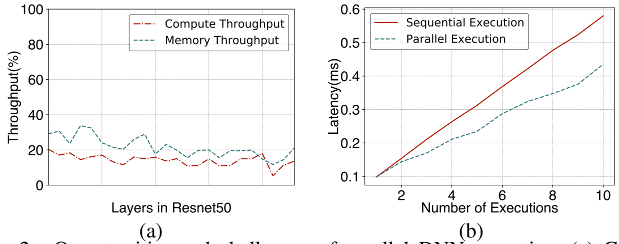


Fig. 2. Opportunities and challenges of parallel DNN execution. (a) GPU utilization with ResNet50. (b) Inference latency of a single fully connected layer using parallelism support available on commodity GPU runtime.

DAGs as multiple streams leads to not only high memory cost, but also latency almost as large as executing these DAGs sequentially (see Fig. 1(f)-(g)). In fact, the state-of-the-art multi-DNN graph rewriters [22], [23] suggest cross-model fusion into a single DAG (see Fig. 1(d)) to comply with the default one-DNN-per-stream execution logic in most deep learning frameworks [4]. Yet these multi-DNN graph rewriters duplicate the shared weights, thus erasing the memory saving of weight-shared DNNs (see Fig. 1(h)).

In this work, we explore graph rewriting strategies dedicated to weight-shared DNNs for efficient multitask inference. Specifically, we aim to generate a single DAG for weight-shared DNNs without duplicating the shared weights (see Fig. 1(e)) to achieve both low latency and memory at runtime when executed on GPU (see Fig. 1(i)). We focus on graph-level optimization to induce minimal changes and dependency to the runtime. Advanced multi-DNN runtime optimizations [5], [7] are often complex to implement and rely on hardware-specific APIs such as CUDA Stream [20], which are inaccessible on platforms like mobile GPUs [24].

We design `MTS`, a novel cross-model graph rewriting framework for efficient multitask inference with weight-shared DNNs. The core of `MTS` is a model stitching algorithm which outputs a single DAG for multiple DNNs without duplicating their shared weights, which minimizes the runtime memory. `MTS` also incorporates a model grouping strategy to organize multiple models in groups to saturate, yet not overwhelm the GPU. Our main contributions and results are as follows.

- We are the first to address the problem of runtime memory saving for cross-model weight sharing.
- We propose `MTS`, which preserves the benefits of cross-model weight sharing to minimize runtime memory usage, and achieves pseudo-parallelism for low latency.
- Experiments are conducted on different hardware platforms, numbers of tasks, network architectures, sharing ratios, batch sizes and heterogeneity. Results show that `MTS` is able to accelerate up to 6.0× comparing to sequentially executing multiple weight-shared DNNs. `MTS` also yields up to 2.5× lower latency and 3.7× less memory usage compared with the SOTA multi-DNN graph rewriter [22].

In the rest of this paper, we state our problem in Sec. II, introduce the `MTS` overview in Sec. III, and elaborate on its model stitching and grouping schemes in Sec. IV and Sec. V, respectively. We present the evaluations in Sec. VI, review related work in Sec. VII, and finally conclude in Sec. VIII.

II. Problem Statement

We focus on graph rewriting of weight-shared DNNs for efficient multitask inference on single-GPU platforms. We justify our objectives and scope in details below.

Objectives. We use *runtime memory* and *overall latency* to assess the efficiency of multitask inference. Specifically, we would like to preserve the memory saving of cross-model weight sharing *i.e.*, the benefits of weight-shared DNNs [3], [6], [8], [9], [18], [19], while achieving low latency when performing multiple inference tasks.

Scope. We target at efficient multitask inference on devices equipped with a single GPU (either desktop- or mobile-grade) by *pseudo-parallelism* at graph-level optimization of DNNs.

- We focus on GPUs because they are common hardware accelerators even on low-resource devices. However, inference with GPU/CPU co-execution [2], [4] is out of our scope.
- We aim at high *parallelism* to execute multiple DNNs for low overall latency. Improving parallelism is a tangible strategy because DNN inference often under-utilizes GPUs due to low computation density of operations and too few

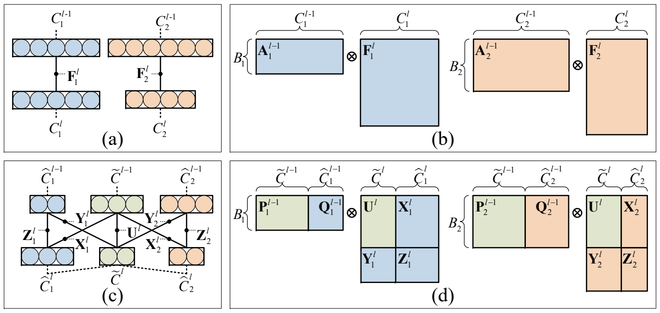


Fig. 3. Notations for two tasks 1 and 2. (a) Layer $(l-1)$ and l without weight-sharing as well as (b) the corresponding dimensions of the activations at layer $(l-1)$ and the weights at layer l. (c) Layer $(l-1)$ and l with weight-sharing as well as (d) the corresponding dimensions of the activations at layer $(l-1)$ and the weights at layer l.

inputs for batching [5], [23], [24], which exists in both desktop- [5] and mobile-grade [24] GPUs. As an example, the NVIDIA GEFORCE RTX 2080 Ti GPU suffers from severe resource under-utilization when executing ResNet50, a representative CNN (see Fig. 2a).

- We resort to implement *pseudo-parallelism* at the *graph* level rather than the *runtime* level, because parallelism APIs for desktop GPUs, such as CUDA Stream [20] and NVIDIA MPS [21], introduce non-trivial contentions [5], scheduling overhead [25], and kernel launch overhead [22], which dominate the overall latency of multitask inference. Even worse, such parallelism APIs are unavailable in mobile GPUs [24]. As a toy example, we measure the latency of executing a single fully connected layer.As shown in Fig. 2b, parallel execution with multiple CUDA streams fails to deliver the expected acceleration over sequential execution.

III. MTS Overview

This section presents the overview of Multi-Task Stitching (MTS), a graph rewriting scheme for low latency, low runtime memory multitask inference on GPU-enabled devices. We illustrate our solution with MTZ [6], [9], a recent method to generate weight-shared DNNs. Our solution also applies to other cross-model weight-sharing schemes [3], [8].

A. Notations

For ease of presentation, we explain our methods with fully-connected (FC) layers and extend to other layers in Sec. IV-B.

Consider T models $\{M_t|1 \leq t \leq T\}$. Each model is a well-trained DNN for an inference task t. Let $\mathbf{F}_t^l$ and $\mathbf{A}_t^l$ be the weight matrix (feature map for CONV layers) and the output activation of layer l ($1 \leq l \leq L$) in model M_t. Then the input activation of layer l is $\mathbf{A}_t^{l-1}$. Accordingly, $\mathbf{A}_t^0$ represents the input of M_t. Furthermore, for M_t, assume B_t is the input batch size and C_t^l is the number of neurons of layer l, so $\mathbf{A}_t^{l-1} \in \mathbb{R}^{B_t \times C_t^{l-1}}$, $\mathbf{F}_t^l \in \mathbb{R}^{C_t^{l-1} \times C_t^l}$.

Without cross-model weight-sharing, the weight matrices $\{\mathbf{F}_t^l\}$ of multiple models $\{M_t\}$ are stored separately (see Fig. 3a for layer l of two models). The corresponding computations are also performed as separate computational graphs, *i.e.*, $\mathbf{A}_t^{l-1} \times \mathbf{F}_t^l$ at layer l for each M_t, as shown in Fig. 3(b).

With cross-model weight-sharing like [6], [9], each model shares certain amount of neurons on a layer basis while keeping other neurons exclusive. Concretely, the weight matrix $\mathbf{F}_t^l$ at layer l of model M_t is split into four portions: $\mathbf{U}^l \in \mathbb{R}^{\widetilde{C}^{l-1} \times \widetilde{C}^l}$, $\mathbf{X}_t^l \in \mathbb{R}^{\widetilde{C}^{l-1} \times \widehat{C}_t^l}$, $\mathbf{Y}_t^l \in \mathbb{R}^{\widehat{C}_t^{l-1} \times \widetilde{C}^l}$, and $\mathbf{Z}_t^l \in \mathbb{R}^{\widehat{C}_t^{l-1} \times \widehat{C}_t^l}$, where $\widetilde{C}^l$, $\widehat{C}_t^l$ are numbers of shared and task-t-exclusive neurons, respectively. Matrix $\mathbf{U}^l$ is shared among models and thus only one copy is stored (see Fig. 3c). Similarly, the input activation $\mathbf{A}_t^{l-1}$ is split into $\mathbf{P}_t^{l-1} \in \mathbb{R}^{B_t \times \widetilde{C}^{l-1}}$ and $\mathbf{Q}_t^{l-1} \in \mathbb{R}^{B_t \times \widehat{C}_t^{l-1}}$. $\mathbf{P}_t^{l-1}$ contains shared neurons of $\mathbf{A}^{l-1}$, while $\mathbf{Q}_t^{l-1}$ contains the remaining exclusive neurons. Fig. 3d illustrates the dimensions of each matrix.

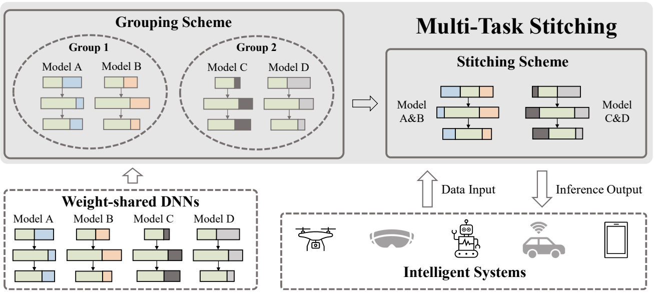


Fig. 4. Workflow of MTS.

Takeaways. As shown in Fig. 3c, cross-model weight-sharing saves storage of weight matrix $\mathbf{U}^l$. However, such storage saving does not readily translate into runtime GPU memory saving due to the lack of a cross-model computational graph rewriter. Naive execution as Fig. 3d duplicates the shared matrix $\mathbf{U}^l$, and regards weight-shared DNNs as separate computational graphs, which may incur high runtime memory footprint and long latency. Our solution is to *stitch* both activations and weight matrices of multiple models as a single computational graph without duplicating the shared matrix $\mathbf{U}^l$ for efficient execution on GPU, as explained in detail next.

B. Functional Modules

MTS aims at pseudo-parallelism of weight-shared DNNs at the graph level. This is realized by stitching weight matrices and activations of multiple tasks into a single stream for better GPU utilization. The core design of MTS consists of two complementary schemes (see Fig. 4).

- A **model stitching** scheme (Sec. IV) that reconstructs multiple computational graphs into a single one without duplicating shared weights, for spatial multiplexing of the GPU among multiple models in parallel.
- A **model grouping** scheme (Sec. V) that determines which models to be grouped for stitching without overwhelming the available resources, where groups of models are sequentially executed with high utilization, *i.e.*, temporal multiplexing the GPU among model groups.

IV. Model Stitching

The model stitching scheme combines the separate computational graphs into a single one without duplicating the shared weights. It demands cross-model stitching of both the input/output activations and the weight matrices. We present the stitching methods for two FC layers (Sec. IV-A), other common layer types (Sec. IV-B), and finally discuss the scalability to stitch more than two models (Sec. IV-C).

A. Stitching Two Fully Connected Layers

Consider the two fully connected (FC) layers in Fig. 3d. Our objective is to reconstruct input activations $\mathbf{A}_1^{l-1}$ and $\mathbf{A}_2^{l-1}$ as $\mathbf{A}_{Stitch}^{l-1}$, and weight matrices $\mathbf{F}_1^l$ and $\mathbf{F}_2^l$ as $\mathbf{F}_{Stitch}^l$, which still results in valid matrix multiplication $\mathbf{A}_{Stitch}^{l-1} \times \mathbf{F}_{Stitch}^l$, without duplicating the shared weights $\mathbf{U}^l$. We start with two special cases which inspire our stitching method for the general case.

1) Special Case 1: Layers l of the two models share all input neurons, *i.e.*, $\widehat{C}_1^{l-1} = \widehat{C}_2^{l-1} = 0$. This is the case when the last layer is fully merged. Accordingly, $C_1^{l-1} = C_2^{l-1} = \widetilde{C}^{l-1}$, $\mathbf{A}_1^{l-1} = \mathbf{P}_1^{l-1}$ and $\mathbf{A}_2^{l-1} = \mathbf{P}_2^{l-1}$. The weight matrices are simplified as $\mathbf{F}_1^l = [\mathbf{U}^l \quad \mathbf{X}_1^l]$ and $\mathbf{F}_2^l = [\mathbf{U}^l \quad \mathbf{X}_2^l]$ (Fig. 5a top).

We can concatenate input activation along the batch-size dimension (Fig. 5a bottom)

$$\mathbf{A}_{Stitch}^{l-1} = \begin{bmatrix} \mathbf{P}_1^{l-1T} & \mathbf{P}_2^{l-1T} \end{bmatrix}^T . \tag{1}$$

For valid matrix multiplication, we can concatenate the weight matrices as (Fig. 5a bottom):

$$\mathbf{F}_{Stitch}^l = [\mathbf{U}^l \quad \mathbf{X}_1^l \quad \mathbf{X}_2^l] . \tag{2}$$

Multiplying the two stitched matrices, we have

$$\mathbf{A}_{Stitch}^l = \mathbf{A}_{Stitch}^{l-1} \times \mathbf{F}_{Stitch}^l = \begin{bmatrix} \mathbf{P}_1^{l-1}\mathbf{U}^l & \mathbf{P}_1^{l-1}\mathbf{X}_1^l & \mathbf{P}_1^{l-1}\mathbf{X}_2^l \\ \mathbf{P}_2^{l-1}\mathbf{U}^l & \mathbf{P}_2^{l-1}\mathbf{X}_1^l & \mathbf{P}_2^{l-1}\mathbf{X}_2^l \end{bmatrix} .$$

Comparing the original calculations without stitching:

$$\mathbf{A}_1^l = \mathbf{A}_1^{l-1} \times \mathbf{F}_1^l = [\mathbf{P}_1^{l-1}\mathbf{U}^l \quad \mathbf{P}_1^{l-1}\mathbf{X}_1^l],$$
$$\mathbf{A}_2^l = \mathbf{A}_2^{l-1} \times \mathbf{F}_2^l = [\mathbf{P}_2^{l-1}\mathbf{U}^l \quad \mathbf{P}_2^{l-1}\mathbf{X}_2^l] .$$

From special case 1, we make the following observations

- We can obtain output activations of the two models from $\mathbf{A}_{Stitch}^l$ with simple matrix rearrangements.
- The stitching strategy introduces certain redundant calculations, *i.e.*, $\mathbf{P}_1^{l-1}\mathbf{X}_2^l$ and $\mathbf{P}_2^{l-1}\mathbf{X}_1^l$.

2) Special Case 2: Layers l of the two models share all output neurons, *i.e.*, $\widehat{C}_1^l = \widehat{C}_2^l = 0$. This may take place when the next layer is fully merged. In this case, we cannot simplify input activations, but weight matrices can be represented by $\mathbf{F}_1^l = \begin{bmatrix} \mathbf{U}^{lT} & \mathbf{Y}_1^{lT} \end{bmatrix}^T$, $\mathbf{F}_2^l = \begin{bmatrix} \mathbf{U}^{lT} & \mathbf{Y}_2^{lT} \end{bmatrix}^T$ (Fig. 5b top). Naturally, we may concatenate weight matrices along the the output-neuron dimension (Fig. 5b bottom) since $C_1^l = C_2^l = \widetilde{C}^l$

$$\mathbf{F}_{Stitch}^l = \begin{bmatrix} \mathbf{U}^{lT} & \mathbf{Y}_1^{lT} & \mathbf{Y}_2^{lT} \end{bmatrix}^T . \tag{3}$$

Stitching of the input activations, however, is slightly more difficult due to the unequal numbers of input neurons. An intuitive solution is to expand $\mathbf{A}_1^{l-1}$ and $\mathbf{A}_2^{l-1}$ with additional zeros. Thus, the stitched input activation is (Fig. 5b bottom).

$$\mathbf{A}_{Stitch}^{l-1} = \begin{bmatrix} \mathbf{P}_1^{l-1} & \mathbf{Q}_1^{l-1} & \mathbf{0} \\ \mathbf{P}_2^{l-1} & \mathbf{0} & \mathbf{Q}_2^{l-1} \end{bmatrix} . \tag{4}$$

The stitched output activation is calculated as

$$\mathbf{A}_{Stitch}^l = \mathbf{A}_{Stitch}^{l-1} \times \mathbf{F}_{Stitch}^l = \begin{bmatrix} \mathbf{P}_1^{l-1}\mathbf{U}^l + \mathbf{Q}_1^{l-1}\mathbf{Y}_1^l \\ \mathbf{P}_2^{l-1}\mathbf{U}^l + \mathbf{Q}_2^{l-1}\mathbf{Y}_2^l \end{bmatrix} .$$

Note that the top half of $\mathbf{A}_{stitch}^l$ is exactly the layer l output activation of M_1, *i.e.*, $\mathbf{A}_1^{l-1} \times \mathbf{F}_1^l$, while the bottom half is exactly the layer l output activation of M_2, *i.e.*, $\mathbf{A}_2^{l-1} \times \mathbf{F}_2^l$.

From special case 2, we make the following observations.

- We can obtain output activations of the two models from $\mathbf{A}_{Stitch}^l$ without redundant calculations.
- Stitching introduces extra zeros in the input activations.

3) General Case: Now we consider the general case to stitch two FC layers. The input activations and the weight matrices are as follows (Fig. 5c top):

$$\mathbf{A}_1^{l-1} = [\mathbf{P}_1^{l-1} \quad \mathbf{Q}_1^{l-1}], \quad \mathbf{A}_2^{l-1} = [\mathbf{P}_2^{l-1} \quad \mathbf{Q}_2^{l-1}],$$
$$\mathbf{F}_1^l = \begin{bmatrix} \mathbf{U}^l & \mathbf{X}_1^l \\ \mathbf{Y}_1^l & \mathbf{Z}_1^l \end{bmatrix}, \quad \mathbf{F}_2^l = \begin{bmatrix} \mathbf{U}^l & \mathbf{X}_2^l \\ \mathbf{Y}_2^l & \mathbf{Z}_2^l \end{bmatrix} .$$

$\mathbf{A}_1^{l-1}$ and $\mathbf{A}_2^{l-1}$ have the same form as those in special case 2, so we can stitch inputs following Eq. (4). As for weight matrices, by taking the matrix size and output values into account carefully, we can deduce from two special cases that

$$\mathbf{F}_{Stitch}^l = \begin{bmatrix} \mathbf{U}^l & \mathbf{X}_1^l & \mathbf{X}_2^l \\ \mathbf{Y}_1^l & \mathbf{Z}_1^l & \mathbf{0} \\ \mathbf{Y}_2^l & \mathbf{0} & \mathbf{Z}_2^l \end{bmatrix} . \tag{5}$$

Accordingly, the stitched output activation is calculated as

$$\mathbf{A}_{Stitch}^l = \mathbf{A}_{Stitch}^{l-1} \times \mathbf{F}_{Stitch}^l = \tag{6}$$
$$\begin{bmatrix} \mathbf{P}_1^{l-1}\mathbf{U}^l + \mathbf{Q}_1^{l-1}\mathbf{Y}_1^l & \mathbf{P}_1^{l-1}\mathbf{X}_1^l + \mathbf{Q}_1^{l-1}\mathbf{Z}_1^l & \mathbf{P}_1^{l-1}\mathbf{X}_2^l \\ \mathbf{P}_2^{l-1}\mathbf{U}^l + \mathbf{Q}_2^{l-1}\mathbf{Y}_2^l & \mathbf{P}_2^{l-1}\mathbf{X}_1^l & \mathbf{P}_2^{l-1}\mathbf{X}_2^l + \mathbf{Q}_2^{l-1}\mathbf{Z}_2^l \end{bmatrix}$$

Discussions. We make the following notes on our scheme to stitch two FC layers, *i.e.*, Eq. (4) and Eq. (5).

- The stitched output activation involves redundant calculations, *i.e.*, $\mathbf{P}_1^{l-1}\mathbf{X}_2^l$ and $\mathbf{P}_2^{l-1}\mathbf{X}_1^l$ in Eq. (6). We quantify the impact of these redundant calculations in Sec. IV-C.
- If we set the redundant elements in $\mathbf{A}_{Stitch}^l$, *i.e.*, $\mathbf{P}_1^{l-1}\mathbf{X}_2^l$ and $\mathbf{P}_2^{l-1}\mathbf{X}_1^l$, to zeros, it just becomes the stitched input activation of the layer $l+1$. That is, we do not need to split and restitch activations between layers.

B. Stitching More Than FC Layers

Now we extend our stitching strategy to layer types in representative convolutional neural networks (CNNs).

1) Stitching Convolutional Layers: Convolutional (CONV) layers are the primary building blocks for CNNs. Since matrix multiplication is a special convolution with input feature map size and kernel size of $(1,1)$, we can stitch CONV layers in the same way as FC layers, which ensures the flexibility and efficiency of `MTS`. The main differences are in data dimensions as the input $\mathbf{A}_t^{l-1}$ and kernel $\mathbf{F}_t^l$ are non-degenerate 4D tensors.

Note that algorithms like `im2col` can convert convolution into matrix multiplication. Hence another solutions is to stitch the FC layers transformed from CONV layers. However, *im2col* may notably increase the memory footprint [26]. As a result, we do not use this solution in the final implementation.

2) Stitching Residual Blocks: Residual blocks are the main components of residual networks. Different from FC and CONV Layers, the residual block is a mixture of multiple layers including several CONV layers and one addition operator, and it's a two-branch network instead of a series of base layers strung sequentially. Existing cross-model weight-sharing methods [9] merge residual blocks by sharing the same

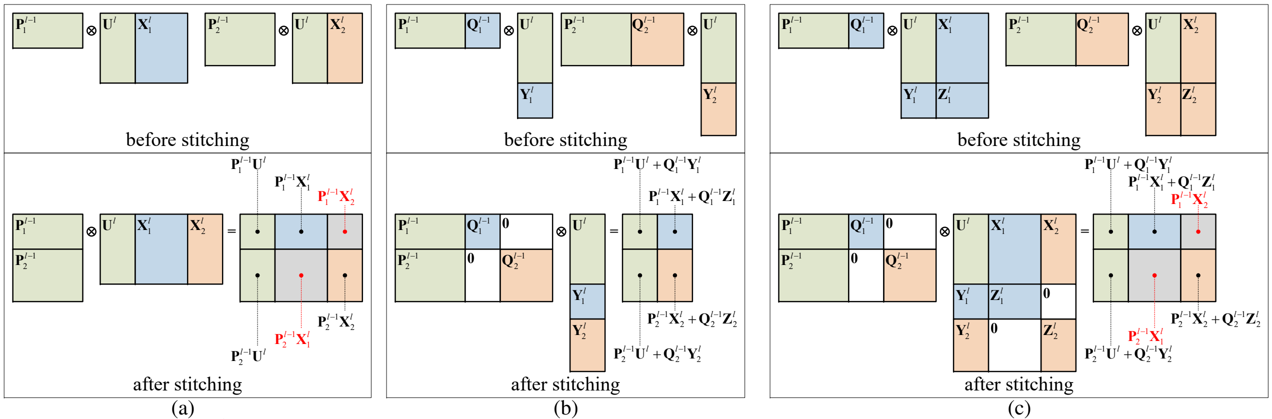


Fig. 5. Illustrations of model stitching for (a) special case 1; (b) special case 2; and (c) the general case.

input/output neurons across these two branches. Therefore, we can stitch CONV layers along each branch independently and exploit the stitched addition operator (more details in the Sec. IV-B3) to combine two branch outputs.

3) Stitching Other Layers/Operators: We briefly discuss the stitching of other common layers or operators in DNNs.

- *Element-wise Binary Operators*. Element-wise binary operators are common in DNN models, *e.g.*, the residual block uses an addition operator to merge branches. Since matrices are only stitched over the neurons dimension, our stitching scheme does not affect element-wise operators. We can directly apply element-wise operators over stitched inputs.
- *Batch Normalization (BN) Layers*. The BN layer is applied to the outputs to keep their mean close to 0 and standard deviation close to 1. Since parameters in the BN layer vary across models, it is impossible to stitch these BN layers into one. However, as the parameters are fixed after training, we can fuse BN layers into their respective previous CONV layers by modifying the weights and bias of CONV layers. Thus stitching BN layers is transformed into stitching CONV layers, which has been solved in Sec. IV-B1.
- *Activation/Pooling Layers*. Activation layers are applied for better fitting non-linear functions, while pooling layers are used to decrease data dimensions. Since the activation function and pooling function take a single element or a cluster of elements as input, our stitching algorithm does not interfere with these layers. The only trifle is that we add additional zeros in inputs and outputs, while activation functions like $Sigmoid(\cdot)$ map zeros to 0.5's. These zeros need to be preserved for correct feed-forward computation.

C. Stitching Multiple Layers

So far we have shown how to stitch layers of two models. We now use the FC layers to illustrate how $T > 2$ models are stitched. We can extend Eq. (4) to stitch input activations and Eq. (5) to stitch weight matrices as:

$$\mathbf{A}_{Stitch}^{l-1} = \begin{bmatrix} \mathbf{P}_1^{l-1} & \mathbf{Q}_1^{l-1} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{P}_T^{l-1} & \mathbf{0} & \dots & \mathbf{Q}_N^{l-1} \end{bmatrix} \tag{7}$$

$$\mathbf{F}_{Stitch}^{l} = \begin{bmatrix} \mathbf{U}^l & \mathbf{X}_1^l & \dots & \mathbf{X}_T^l \\ \mathbf{Y}_1^l & \mathbf{Z}_1^l & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{Y}_T^l & \mathbf{0} & \dots & \mathbf{Z}_T^l \end{bmatrix} \tag{8}$$

Hence $\mathbf{A}_{Stitch}^{l}$, the result of $\mathbf{A}_{Stitch}^{l-1} \times \mathbf{F}_{Stitch}^{l}$, has a similar structure to $\mathbf{A}_{Stitch}^{l-1}$, from which we can obtain $\mathbf{A}_t^l$.

Analysis of Stitched Model. As mentioned in Sec. IV-A, the stitching scheme introduces redundant computation. We now estimate the overall computation of a stitched model.

Assume $\mathbf{A}_{Stitch}^{l}$ and $\mathbf{F}_{Stitch}^{l}$ have dimensions of $B_S \times C_S^{l-1}$ and $C_S^{l-1} \times C_S^l$. Then B_S, C_S^{l-1}, and C_S^l are computed as:

$$B_S = \sum_{t=1}^{T} B_t, \qquad C_S^{l-1} = \widetilde{C}^{l-1} + \sum_{t=1}^{T} \widehat{C}_t^{l-1}, \qquad C_S^l = \widetilde{C}^l + \sum_{t=1}^{T} \widehat{C}_t^l .$$

The floating point operations (FLOPs) of the stitched model is estimated as $G_S^l = B_S \times C_S^{l-1} \times C_S^l$. For simplicity, assume $\forall t \in [1,T], B = B_t, \widetilde{C}_t^{l-1} = \alpha^{l-1}C^{l-1}, \widetilde{C}_t^l = \alpha^l C^l$, where $\alpha^{l-1}, \alpha^l \in [0,1]$ are the sharing ratios in cross-model weight-sharing [6], [9]. Substituting with Eq. (6), the stitched weight/model size (MS) and total FLOPs can be estimated as

$$\begin{aligned} \text{MS} &= C_S^{l-1} \times C_S^l \\ &= \left(\alpha^{l-1} + T \cdot (1-\alpha^{l-1})\right)\left(\alpha^l + T \cdot (1-\alpha^l)\right) \cdot C^{l-1}C^l \\ \text{FLOPs} &= B_S \times C_S^{l-1} \times C_S^l \\ &= T\left(\alpha^{l-1} + T \cdot (1-\alpha^{l-1})\right)\left(\alpha^l + T \cdot (1-\alpha^l)\right) \cdot C^{l-1}C^l B \end{aligned}$$

Hence the stitched model size is $\mathcal{O}(T^2)$ with $\mathcal{O}(T^3)$ FLOPs.

Discussions. We make the following notes.

- Since the GPU is under-utilized [5], [24] for inference, the stitching scheme improves the throughput and reduces the latency of multitask inference, as empirically validated in Sec. VI (`MTS` against `Sequential`).
- The redundant computation does impair the scalability to more *e.g.*, tens of tasks, which motivates the model grouping design in Sec. V for temporal multiplexing the GPU without overwhelming. Experimental results show that with

Algorithm 1: Model Grouping Algorithm

```
   input : weight-shared models $M_t (1 \le t \le T)$,
           corresponding inputs $A_t^0$
   output: model grouping scheme $\mathcal{G}$, s.t. $\forall M_t \in \{M_t \mid 1 \le t \le T\}$, $\exists! \mathcal{G}_i \in \mathcal{G}$, $M_t \in \mathcal{G}_i$
1  if models have the same structure and input batch size then
2      $\mathcal{F}_0 \leftarrow 0$;
3      foreach $n \in [2, T]$ do
4          $L_n \leftarrow$ latency to assign $n$ models in one group;
5          $\mathcal{F}_n \leftarrow \min_{1 \le j \le n} \{L_n,\ \mathcal{F}_{n-i} + L_i\}$ ;
6      extract $\mathcal{G}$ from the derivation of $\mathcal{F}_T$;
7  else if $T$ is small then
8      $\mathcal{H}_\emptyset \leftarrow 0$;
9      foreach $\mathcal{S} \subseteq \{M_t \mid 1 \le t \le T\}$ do
10         $L_\mathcal{S} \leftarrow$ latency to assign $\mathcal{S}$ models in one group;
11         $\mathcal{H}_\mathcal{S} \leftarrow \min_{\mathcal{S}' \subset \mathcal{S}} \{L_\mathcal{S},\ \mathcal{H}_{\mathcal{S}'} + \mathcal{H}_{\mathcal{S} \setminus \mathcal{S}'}\}$;
12     extract $\mathcal{G}$ from the derivation of $\mathcal{H}_{\{M_t \mid 1 \le t \le T\}}$;
13 else
14     foreach $G \in [1, T]$ do
15         construct empty groups $\hat{\mathcal{G}} \leftarrow \{\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_G\}$;
16         set group-queue $[\mathcal{G}_1, \cdots, \mathcal{G}_G, \mathcal{G}_G, \cdots, \mathcal{G}_1, \cdots]$;
17         sort models in descending order of latency;
18         group models in the order of group-queue;
19         get $\hat{\mathcal{G}}$'s latency $L_{\hat{\mathcal{G}}}$, update $\mathcal{G}$ if $L_{\hat{\mathcal{G}}} < L_\mathcal{G}$;
20 return $\mathcal{G}$
```

model grouping, our method achieves comparable latency with NETFUSE [22], the state-of-the-art multi-DNN graph rewriter, with notably lower runtime GPU memory usage.

- Note that MS and FLOPs decrease as α^{l-1} and α^l increase. It indicates that when sharing ratios comes to 1, *i.e.*, all T models are exactly the same, `MTS` simply batches inputs. Therefore, `MTS` is a novel extension of input batching.

V. Model Grouping

In this section, we present the model grouping scheme of `MTS`. It facilitates efficient multitask inference with even tens of models. Specifically, the T models are organized into G groups $\mathcal{G} = \{\mathcal{G}_i \mid 1 \le i \le G\}$. The groups are executed sequentially, whereas models within a group are stitched and executed in parallel. For efficient grouping, we propose a greedy-based grouping scheme as illustrated in Algorithm 20. Its details are explained below.

- **Same Model Structure and Batch Size (Line 1-6)**. In this case, each model can be considered as the same inference task. Thus the latency of each group only depends on the number of tasks, and we can use dynamic programming to get the optimal grouping. Specifically, let $\mathcal{F}_n$ as the minimum latency of the first n models. The recursive definition of $\mathcal{F}_n$ in line 5 chooses the optimal scheme by assigning j tasks to one group, and the remaining $n - j$ tasks to other groups. Finally, we construct an optimal group scheme according to the computing progress of $\mathcal{F}_T$ (line 6).
- **Small Number of Tasks T (Line 7-12)**. In this case, we use state compression dynamic programming to get the optimal grouping scheme. Let $\mathcal{H}_\mathcal{S}$ be the minimum latency of all models in set $\mathcal{S}$. The recursive definition (line 11) and the optimal grouping scheme construction (line 12) are identical to the previous case, with the exception that the number of tasks n is replaced with the group set $\mathcal{S}$.
- **General Case (Line 14-19)**. In the most general case, we group models greedily: we enumerate the number of groups from 1 to T, and put models into groups in a balanced way (Line 15-18). The best grouping scheme is chosen. The criterion is to balance the latency among groups.

Discussions. We make two notes on the model grouping.

- Despite the greedy grouping, it finds the optimal solution if the model sizes and batch sizes are the same, or the number of tasks is small (empirically set as 15).
- The algorithm requires inference latency (Line 4, 10, 17), which can be estimated offline by direct measurements.

VI. Evaluation

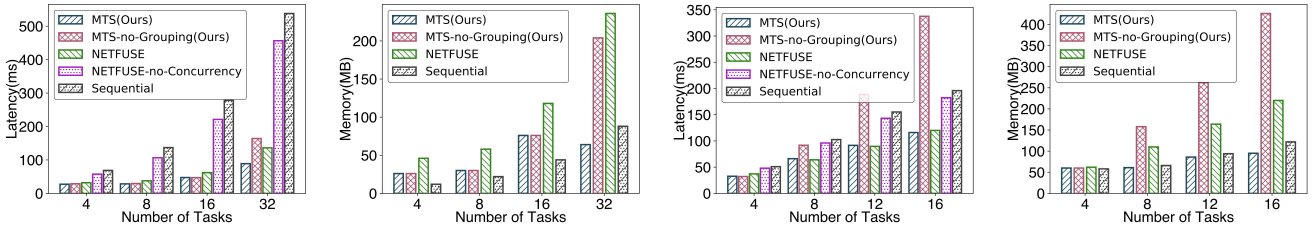
A. Evaluation Setup

Platforms and Implementation. We conduct experiments on two platforms: Jetson TX2 and an edge server. Jetson TX2 is a mobile computing platform equipped with a Quad-Core ARM Cortex-A57 MPCore (NVIDIA Denver 2 CPU was disabled during experiments) and 8GB RAM, as well as a 256-core Pascal-based GPU. The edge server is equipped with an 32-core Intel Xeon E5-2620@2.10GHz processor and 256GB RAM, as well as a NVIDIA GEFORCE RTX 2080 Ti. All the algorithms are implemented with PyTorch in Python.

Inference Workload. We experiment with three representative CNNs: ResNet18, ResNet50 and VGG16. Due to the limited resources on mobile and edge devices, we also considered pruned versions of these models. Note that cross-model weight-sharing schemes [6], [9] can also merge pruned models. To generate multitask inference workload, we merge a given number of the three CNN types, either pruned or unpruned, by MTZ [6]. We test different model number, pruning ratios, and sharing ratios. We consider a batch size of 1 since most inference tasks demand real-time processing. The detailed configurations are deferred to each experiment.

Baselines. We compare `MTS` with the following methods.

- `Sequential`: It selects one model from all in a round-robin fashion and performs the inference one by one.
- `NETFUSE` [22]: It is the state-of-the-art multi-DNN graph rewriter. It leverages operations like group convolution and batch matrix multiplication for cross-model fusion.
- `NETFUSE-no-Concurrency`: It is the original `NETFUSE` with multi-stream execution disabled. It is to simulate the mobile GPU runtime because multi-stream execution is not supported by most mobile GPUs [24].
- `MTS-no-Grouping`: It is `MTS` without the grouping.



(a) Latency (ResNet18) (b) Memory (ResNet18) (c) Latency (VGG16) (d) Memory (VGG16)

Fig. 6. Performance comparisons on mobile platforms (Jetson TX2).

Metrics. We use overall inference latency and peak memory footprint to assess the performance of each algorithm. The memory footprint of `NETFUSE-no-Concurrency` is omitted because it has the same result as `NETFUSE`. We measure the inference latency on GPU by calculating the average of 500 inference latency after 2 warmups, and capture the peak memory footprint during execution with NVIDIA Nsight Systems [27]. Since the reconstruction of weights induces extra delay in inference, the models in `Sequential`, `NETFUSE`, and `NETFUSE-no-Concurrency` are not merged by cross-model weight-sharing schemes for fair comparison.

B. Evaluation Results

We now present our evaluation results with various platforms and workload settings.

1) Performance on Mobile Devices: In this experiment, we compare different algorithms on Jetson TX2.

Settings. We choose pruned ResNet18 and VGG16 as the model types. Each model has 90% neurons pruned (pruning ratio = 0.9) and all of them share 90% of the remaining neurons with each other (sharing ratio = 0.9). In this experiment, we set the batch size of all models to 1 and vary the number of weight-shared models from 4 to 32.

Results. Fig. 6 shows the inference latency and peak memory footprint of ResNet18 and VGG16 on Jetson TX2.

For ResNet18 (see Fig. 6a and Fig. 6b), `MTS` yields about $2.5\times$, $4.8\times$, $5.8\times$, $6.0\times$ speedup against `Sequential` when executing 4, 8, 16, 32 models, respectively. `MTS`, however, consumes more memory than `Sequential` when there are 4, 8 or 16 models. This is caused by the the temporary runtime memory dominating in total memory usage. When the number of tasks reaches 32, `MTS`'s memory footprint is reduced significantly and surpasses `Sequential` ($1.4\times$ memory saving), benefiting from the grouping scheme. Compared to `NETFUSE`, `MTS` is about $1.4\times$ faster. The latency gap is closing, but `NETFUSE` consumes much more memory: $1.8\times$, $1.9\times$, $1.6\times$ and $3.7\times$, respectively. As for `MTS-no-Grouping`, when the number of tasks is small (4, 8 or 16), no grouping is activated and it leads to the same performance as `MTS`. When the number grows to 32, all tasks are grouped into three, thus `MTS` saves $1.84\times$ latency and $3.18\times$ memory usage. At last, the latency of `NETFUSE-no-Concurrency` is almost the same as `Sequential` ($2.1\times$, $3.7\times$, $4.7\times$ and $5.1\times$ slower than `MTS`, respectively), indicating that `NETFUSE` is unfit for mobile platforms without multi-stream APIs.

For VGG16 (see Fig. 6c and Fig. 6d), `MTS` achieves $1.5\times$ speedup and $1.3\times$ memory saving than `Sequential`. Compared with the speedup for ResNet18, the lower speedup is due to VGG16 being more computational intensive. For the same reason, the latency of `MTS-no-Grouping` even exceeds that of `Sequential`. The memory footprint of `MTS-no-Grouping` is also drastically larger than that of `NETFUSE` since the all-in-one stitched VGG16 model consumes large amounts of runtime memory.

In a word, `MTS` achieves the best latency-memory balance than baselines. For comparison, `MTS` accelerates up to $6.0\times$ and saving $1.4\times$ memory compared to `Sequential`. As for `NETFUSE`, `MTS` is $1.5\times$ faster and $3.7\times$ memory saving. Furthermore, the results of `MTS-no-Grouping` demonstrate the necessity of model grouping.

2) Performance on Edge Servers: In this experiment, we test the algorithms on a desktop-grade GPU for edge servers.

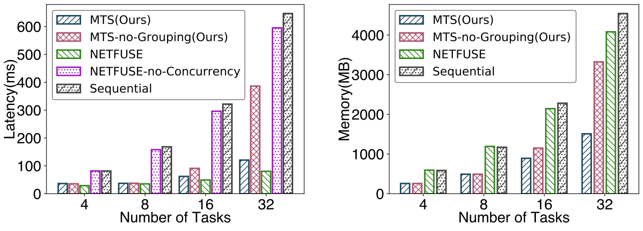
Settings. We use a server equipped with an RTX 2080 Ti GPU as the platform, and ResNet50 as the model type. Due to the adequate resource on the desktop-grade GPU, we directly adopt the unpruned ResNet50 as the model for inference tasks. The sharing ratio and batch size are 0.9 and 1, respectively.

Results. Fig. 7 plots the inference latency and peak memory footprint of each method when executing 4 to 32 weight-shared ResNet50 models on the edge server. As with the results on Jetson TX2, `MTS` is $5.3\times$ faster than `Sequential`. However, `MTS` is slightly slower than `NETFUSE`. This may stem from the larger bandwidth and more CUDA cores, allowing faster execution of group convolution and batch matrix multiplication used by `NETFUSE`. In terms of memory usage, `MTS` still notably outperforms `NETFUSE`, consuming only 37% of memory `NETFUSE` does. `Sequential`'s peak memory footprint grows drastically because the pruning ratio is 1 and it is the models' parameters rather than temporary runtime memory that dominates in the total memory footprint.

3) Impact of Sharing Ratios: Another hyperparameter of the inference workload is the sharing ratio. A large sharing ratio is set for high task relatedness or limited resources [6].

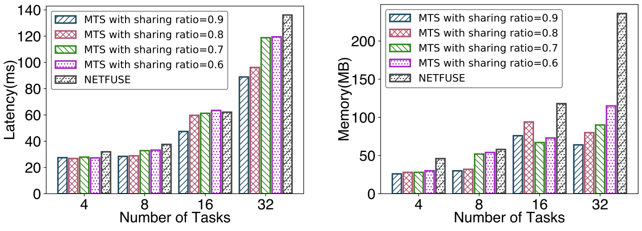
Settings. We use Jetson TX2 as the platform, the pruned ResNet18 with a pruning ratio of 0.9 as the model type and set the batch size to 1. We experiment with four sharing ratios (0.9, 0.8, 0.7, 0.6) and four numbers of tasks (4, 8, 16, 32).

Results. Fig. 8 shows the inference latency and peak memory footprint of ResNet18 with different sharing ratios. When operating the same grouping scheme (number of tasks is 4, 8 or 32), the inference latency and the memory footprint of `MTS` decrease with the increase of sharing ratio, which is consistent with our analysis in Sec. IV-C. `MTS` achieves the



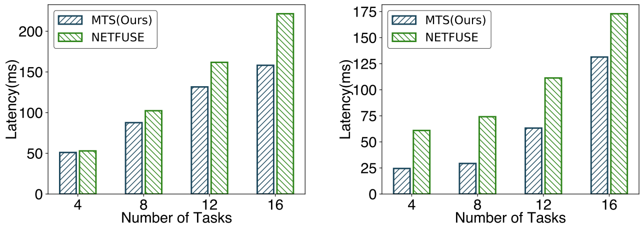
(a) Latency (ResNet50) (b) Memory (ResNet50)

Fig. 7. Performance comparisons on edge servers.



(a) Latency (b) Memory

Fig. 8. Impact of sharing ratios using ResNet18 models on Jetson TX2.



(a) Pruning ratios=0.9/0.85 (b) Batch sizes=1/2/3

Fig. 9. Impact of model heterogeneity. (a): latency of VGG16's with mixed pruning ratios; (b) latency of ResNet18's with mixed batch sizes.

lowest latency and memory cost with all sharing ratios. The gain is more notable with more models *e.g.*, 32. This is because the model grouping scheme is activated by an excessively high computing need. Then both the inference latency and peak memory will be greatly reduced by model grouping of MTS.

4) Impact of Heterogeneity in Inference Workload: In this experiment, we test the impact of heterogeneity in inference workload on the performance of MTS. We consider two types of heterogeneity: mixed layer widths and input batch sizes.

Settings. We conduct two experiments on Jetson TX2.

- In the first experiment, we force MTS and NETFUSE to combine multiple VGG16's compressed with two different pruning ratios: 0.9 and 0.85. MTS naturally functions with unequal layer widths. For NETFUSE to work with unequal layer widths, it extends the narrower layers with redundant neurons. The sharing ratio is 0.9 and the batch size is 1.
- In the second experiment, we vary the batch size from 1 to 3 and use ResNet18 as the model type. The stitching strategy of MTS is designed to handle different batch sizes. Nevertheless, NETFUSE needs to group these models according to the batch size and execute them one by one. We set both the pruning ratio and the sharing ratio to 0.9 .

Results. Fig. 9a shows that the inference latency of VGG16s with mixed pruning ratios. Comparing Fig. 9a with Fig. 6c, where a fixed pruning ratio of 0.9 is adopted for all VGG16 models, the gain in latency of MTS over NETFUSE becomes more notable, because NETFUSE has to pad redundant neurons for combining models of different widths. Fig. 9b shows that the inference latency of ResNet18s with mixed batch sizes. Compared with Fig. 6a, MTS is much more efficient than NETFUSE, especially when number of tasks is relatively small, *e.g.*, 4, 8. The forcing sequential execution of NETFUSE leads to severe resource under-utilization.

5) Summary of Experimental Results: We summarize our main experimental findings as follows.

- MTS can accelerate Sequential up to 6.0 times.
- With the model grouping scheme, MTS saves 3.2× latency and 4.5× memory footprint over MTS-no-Grouping.
- Overall, MTS outperforms NETFUSE, a state-of-the-art multi-DNN graph rewriter, in both inference latency (up to 1.5×) and runtime memory (over 3.7×). The advantages of MTS are more notable with heterogeneity in inference workload, achieving up to 2.5× speedup.

VII. Related Work

Our work is related to the following categories of research.

A. Weight-Shared DNNs

Cross-model weight sharing facilitates DNN deployment to low-memory devices. It applies to DNNs for either a *single* task or *multiple* tasks. In single-task weight-shared DNNs, each DNN is often a model variant for the same inference task, yet of a different complexity-accuracy trade-off. Examples include early-exits [28], nested architectures [1], etc. In multitask weight-shared DNNs, each DNN is trained for a different inference task. Multitask weight sharing is feasible for correlated tasks, and can be achieved by cross-model quantization [18] or fine-tuning [3], [6], [8], [9]. We apply MTZ [6] to generate weight-shared DNNs, for it allows adaptive weight sharing and supports diverse layer types. Other studies either enforce full weight sharing [3] or support convolutional layers only [8].

Although weight-shared DNNs save storage, their execution may not speed up for multitask inference. Naive execution of such networks results in multiple data flows [29]. This leads to the same delay as running unshared DNNs sequentially, and erases the memory saving of weight sharing. MTS is the first attempt at efficient weight-shared DNN execution for multitask inference while retaining the memory saving.

B. Multi-DNN Graph Rewriting

Deep learning(DL) frameworks such as TensorFlow and PyTorch represent DNNs as computational DAGs. Graph rewriters such as TVM [11] and NVDIA TensorRT [12] apply configured graph substitution rules to output mathematically equivalent DAGs that run faster on the given hardware platform. Yet these rewriters optimize each DAG in isolation and are ineffective for multi-DNN graph rewriting [22], [23].

HiveMind [23] and NETFUSE [22] are two SOTA multi-DNN graph rewriters. The idea is to perform cross-model layer fusion to increase the computational intensity of operations, and thus the GPU utilization [23]. HiveMind assumes the same input for weight-shared DNNs, while NETFUSE supports cross-model layer fusion of DNNs with different inputs and outputs. However, neither HiveMind nor NETFUSE retains the memory saving of weight sharing, and they pose constraints such as the same channel or input size on the DNNs. These drawbacks motivates the model stitching strategy in our MTS.

C. Multi-DNN Runtime Scheduling

Given DAGs as input, a multi-DNN runtime schedules the DAG operations to maximize the pipelined or parallel execution on the targeting platform. Despite multi-tenancy APIs such as CUDA stream [20] and NVIDIA MPS [21], multi-DNN runtime scheduling is still challenging because most DL frameworks adopts one-DNN-per-process execution model by default [4]. DeepEye [2] interleaves the executions of CONV and FC layers of multiple DNNs for latency hiding. DART [4] is a pipelined multi-DNN scheduling framework under real-time constraints. MASA [7] is the latest memory-aware multi-DNN runtime scheduler. For multi-DNN runtime scheduling on mobile GPU, ParallelFusion [24] proposes kernel fusion to maximum the utilization of mobile GPU.

Our MTS is complementary. We use a single CUDA stream as the runtime for weight-shared DNNs to avoid the API's inefficient parallelism support [5], [23], [25] and because current mobile GPUs only allows a single stream [24].

VIII. Conclusion

In this paper, we introduce MTS, a graph rewriting framework for efficient multitask inference with weight-shared DNNs. MTS uses a model stitching scheme to output a single DAG for multiple DNNs with shared weights. The runtime memory usage is minimized via avoiding the duplication of shared weights. With the help of a dedicated model grouping strategy, it also achieves a near optimal runtime latency. We conducted extensive experiments on different hardware platforms, numbers of tasks, network architectures, sharing ratios, batch sizes and model heterogeneity. Results show that MTS accelerates up to 6.0× compare to sequentially executing multiple weight-shared DNNs. MTS also yields up to 2.5× lower latency and 3.7× less memory usage compared with NETFUSE, a state-of-the-art multi-DNN graph rewriter. We envision our work as a critical step towards the full-stack optimization for efficient multi-DNN execution.

Acknowledgement

This research was supported in part by the National Key R&D Program of China under grant 2018YFB2100300, the NSFC under grant 62072272.

This research was also supported by the Lee Kong Chian Fellowship awarded to Zimu Zhou by Singapore Management University. Zimu Zhou is the corresponding author.

References

[1] B. Fang, X. Zeng, and M. Zhang, "Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision," in *MobiCom*, 2018.

[2] A. Mathur, N. D. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar, "Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware," in *:MobiSys*, 2017.

[3] S. Lee and S. Nirjon, "Fast and scalable in-memory deep multitask learning via neural weight virtualization," in *MobiSys*, 2020.

[4] Y. Xiang and H. Kim, "Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference," in *RTSS*, 2019.

[5] F. Yu, S. Bray, D. Wang, L. Shangguan, X. Tang, C. Liu, and X. Chen, "Automated runtime-aware scheduling for multi-tenant dnn inference on gpu," in *ICCAD*, 2021.

[6] X. He, Z. Zhou, and L. Thiele, "Multi-task zipping via layer-wise neuron sharing," in *NeurIPS*, 2018.

[7] B. Cox, J. Galjaard, A. Ghiassi, R. Birke, and L. Y. Chen, "Masa: Responsive multi-dnn inference on the edge," in *PerCom*, 2021.

[8] C.-E. Wu, J.-H. Lee, T. S. Wan, Y.-M. Chan, and C.-S. Chen, "Merging well-trained deep cnn models for efficient inference," in *APSIPA*, 2020.

[9] X. He, X. Wang, Z. Zhou, J. Wu, Z. Yang, and L. Thiele, "On-device deep multi-task inference via multi-task zipping," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2021.

[10] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: a comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[11] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *OSDI*, 2018.

[12] NVIDIA, "NVIDIA TensorRT," https://developer.nvidia.com/tensorrt.

[13] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "Gpu scheduling on the nvidia tx2: Hidden details revealed," in *RTSS*, 2017.

[14] D. Gao, X. He, Z. Zhou, Y. Tong, K. Xu, and L. Thiele, "Rethinking pruning for accelerating deep inference at the edge," in *KDD*, 2020.

[15] 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 *IPSN*, 2016.

[16] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *MobiSys*, 2018.

[17] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications," in *MobiSys*, 2017.

[18] Y.-M. Chou, Y.-M. Chan, J.-H. Lee, C.-Y. Chiu, and C.-S. Chen, "Unifying and merging well-trained deep neural networks for inference stage," in *IJCAI*, 2018.

[19] T. S. Wan, J.-H. Lee, Y.-M. Chan, and C.-S. Chen, "Co-compressing and unifying deep cnn models for efficient human face and speaker recognition," in *CVPR Workshops*, 2019.

[20] NVIDIA, "NVIDIA CUDA streams," https://developer.download.nvidia.com/CUDA/training/StreamsAnd ConcurrencyWebinar.pdf.

[21] ——, "NVIDIA Multi-Process Service," https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[22] J. S. Jeong, S. Kim, G.-I. Yu, Y. Lee, and B.-G. Chun, "Accelerating multi-model inference by merging dnns of different weights," *arXiv preprint arXiv:2009.13062*, 2020.

[23] D. Narayanan, K. Santhanam, A. Phanishayee, and M. Zaharia, "Accelerating deep learning workloads through efficient multi-model execution," in *NeurIPS Workshops*, 2018.

[24] J. Lee, Y. Liu, and Y. Lee, "Parallelfusion: Towards maximum utilization of mobile gpu for dnn inference," in *EMDL*, 2021.

[25] W. Kwon, G.-I. Yu, E. Jeong, and B.-G. Chun, "Nimble: Lightweight and parallel gpu task scheduling for deep learning," in *NeurIPS*, 2020.

[26] Z. Ji, "Ilp-m conv: Optimize convolution algorithm for single-image convolution neural network inference on mobile gpus," *arXiv preprint arXiv:1909.02765*, 2019.

[27] NVIDIA, "NVIDIA Nsight Systems," https://developer.nvidia.com/nsight-systems.

[28] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama, "Adaptive neural networks for efficient inference," in *ICML*, 2017.

[29] X. He, D. Gao, Z. Zhou, Y. Tong, and L. Thiele, "Pruning-aware merging for efficient multitask inference," in *KDD*, 2021.