TaskFusion: An Efficient Transfer Learning Architecture with Dual Delta Sparsity for Multi-Task Natural Language Processing

Zichen Fan
University of Michigan
Ann Arbor, Michigan, USA
zcfan@umich.edu

Qirui Zhang
University of Michigan
Ann Arbor, Michigan, USA
qiruizh@umich.edu

Pierre Abillama
University of Michigan
Ann Arbor, Michigan, USA
pabillam@umich.edu

Sara Shoouri
University of Michigan
Ann Arbor, Michigan, USA
sshoouri@umich.edu

Changwoo Lee
University of Michigan
Ann Arbor, Michigan, USA
cwoolee@umich.edu

David Blaauw
University of Michigan
Ann Arbor, Michigan, USA
blaauw@umich.edu

Hun-Seok Kim
University of Michigan
Ann Arbor, Michigan, USA
hunseok@umich.edu

Dennis Sylvester
University of Michigan
Ann Arbor, Michigan, USA
dmcs@umich.edu

ABSTRACT

The combination of pre-trained models and task-specific fine-tuning schemes, such as BERT, has achieved great success in various natural language processing (NLP) tasks. However, the large memory and computation costs of such models make it challenging to deploy them in edge devices. Moreover, in real-world applications like chatbots, multiple NLP tasks need to be processed together to achieve higher response credibility. Running multiple NLP tasks with specialized models for each task increases the latency and memory cost latency linearly with the number of tasks. Though there have been recent works on parameter-shared tuning that aim to reduce the total parameter size by partially sharing weights among multiple tasks, computation remains intensive and redundant despite different tasks using the same input. In this work, we identify that a significant portion of activations and weights can be reused among different tasks, to reduce cost and latency for efficient multi-task NLP. Specifically, we propose *TaskFusion*, an efficient transfer learning software-hardware co-design that exploits delta sparsity in both weights and activations to boost data sharing among tasks. For training, *TaskFusion* uses ℓ_1 regularization on delta activation to learn inter-task data redundancies. A novel hardware-aware sub-task inference algorithm is proposed to exploit the dual delta sparsity. We then designed a dedicated heterogeneous architecture to accelerate multi-task inference with an optimized scheduling to increase hardware utilization and reduce off-chip memory access. Extensive experiments demonstrate that *TaskFusion* can reduce the number of floating point operations (FLOPs) by over 73% in multi-task NLP with negligible accuracy loss, while adding a new task at the cost of only $< 2\%$ parameter size increase. With the proposed architecture and optimized scheduling, *TaskFusion* can achieve 1.48-2.43× performance and 1.62-3.77× energy efficiency than those using state-of-the-art single-task accelerators for multi-task NLP applications.

ISCA '23, June 17–21, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589040

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**.

KEYWORDS

transfer learning, multi-task, sparsity, transformer, heterogeneous architecture, natural language processing, accelerator, deep learning

ACM Reference Format:
Zichen Fan, Qirui Zhang, Pierre Abillama, Sara Shoouri, Changwoo Lee, David Blaauw, Hun-Seok Kim, and Dennis Sylvester. 2023. TaskFusion: An Efficient Transfer Learning Architecture with Dual Delta Sparsity for Multi-Task Natural Language Processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579371.3589040

1 INTRODUCTION

Natural language processing (NLP) is critical in enabling devices to learn and act appropriately by understanding language with context, which is becoming a new interface between users and systems such as voice assistants and chatbots. Recently, transfer learning [1, 30] has become one of most trending approaches for NLP. One of the transfer learning technique is model pre-training and fine-tuning, which has been shown by previous studies [3, 7, 24, 34] to be effective for improving various NLP tasks. The idea is to first pre-train a generalized model, then fine-tune its parameters for a specific sub-task such that the same network structure is reused. Although that task-specific fine-tuning strategy drastically reduces training complexity, the computational and storage requirements become bottlenecks when the system needs inferences for multiple tasks.

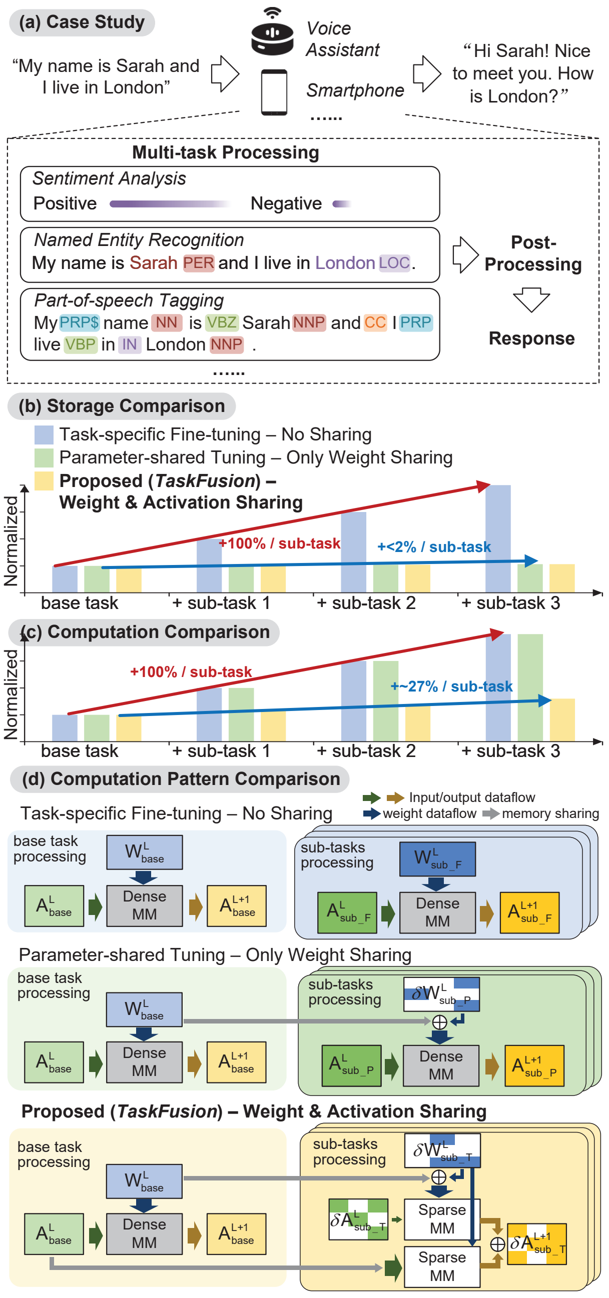
(a) Case Study
"My name is Sarah and I live in London"
Voice Assistant
Smartphone
"Hi Sarah! Nice to meet you. How is London?"
Multi-task Processing
Sentiment Analysis
Positive
Negative
Named Entity Recognition
My name is Sarah PER and I live in London LOC.
Part-of-speech Tagging
My PRP$ name NN is VBZ Sarah NNP and CC I PRP live VBP in IN London NNP .
Post-Processing
Response
(b) Storage Comparison
Task-specific Fine-tuning – No Sharing
Parameter-shared Tuning – Only Weight Sharing
Proposed (TaskFusion) – Weight & Activation Sharing
Normalized
+100% / sub-task
+<2% / sub-task
base task
+ sub-task 1
+ sub-task 2
+ sub-task 3
(c) Computation Comparison
+100% / sub-task
+~27% / sub-task
(d) Computation Pattern Comparison
Input/output dataflow
weight dataflow
memory sharing
Task-specific Fine-tuning – No Sharing
base task processing
sub-tasks processing
Dense MM
Parameter-shared Tuning – Only Weight Sharing
Proposed (TaskFusion) – Weight & Activation Sharing
Sparse MM

Figure 1: TaskFusion Overview: (a) A case study of multi-task processing in edge devices.(b) Storage requirement versus number of sub-tasks comparison (c) Computation consumption versus number of sub-tasks comparison. (d) Computation pattern comparison. under multi-task processing scenario.

Consider a practical scenario of a smart voice assistant where different NLP tasks need to be executed on one sentence or a sentence group (sentence batch) as shown in Fig. 1 (a). In this scenario, the same sentence is processed for multiple tasks such as sentiment analysis [41], named entity recognition [38], part-of-speech tagging[27] to understand the user's request and give a satisfactory answer. In the proposed framework, one task is treated as the base task and the other tasks as sub-tasks. Fig. 1 (b) shows the storage requirement of edge devices versus the number of sub-tasks when multi-task NLP is based on conventional task-specific fine-tuning, where each model for a sub-task requires a new set of weights sizing the same as the pre-trained model. Therefore, 100% extra storage per task is needed. When $\text{BERT}_{\texttt{large}}$ (340 million parameters [7]) is used as the pre-trained model, an additional ≈ 1.3GB of memory is needed per sub-task, which is impractical for resource-constrained edge devices.

In order to alleviate the storage bottleneck for multi-task scenarios, parameter-shared tuning [13, 19, 52] has recently been proposed. Instead of fine-tuning all parameters in the pre-trained model, parameter-shared tuning only changes a small amount of weights for each sub-task and leaves the rest unchanged from the pre-trained model. Fig. 1 (b) shows the storage requirement versus the number of tasks for parameter-shared tuning, which increases linearly with the number of tasks but with a much smaller slope compared to task-specific fine-tuning. However, parameter-shared tuning only reduces the amount of storage but does not impact the amount of computation. Fig. 1 (c) shows the computation comparison under the single-input-multi-task scenario with conventional task-specific fine-tuning and parameter-shared tuning approach expressed in the following formulas where A^l and W^l denote activations and weights of the l_{th} linear layer.

Layer L of the base task:

$$A^{L+1}_{base} = A^{L}_{base} \times W^{L}_{base} \tag{1}$$

Layer L of a sub-task with task-specific fine-tuning:

$$A^{L+1}_{sub\_F} = A^{L}_{sub\_F} \times W^{L}_{sub\_F} \tag{2}$$

Layer L of a sub-task with parameter-shared tuning:

$$\begin{aligned} A^{L+1}_{sub\_P} &= A^{L}_{sub\_P} \times W^{L}_{sub\_P} \\ &= A^{L}_{sub\_P} \times (W^{L}_{base} + \delta W^{L}_{sub\_P}) \end{aligned} \tag{3}$$

For a linear layer L, Fig. 1 (d) compares the computation patterns for the base task and sub-tasks with task-specific fine-tuning or parameter-shared tuning. The base task typically corresponds to the pre-trained model. Suppose activations and weights are represented as dense matrices. Then for the base task, always one dense matrix-matrix multiplication is executed. For task-specific fine-tuning, W_{sub_F} is completely different from W_{base} and a dense matrix-matrix multiplication is still needed in each sub-task. Therefore, task-specific fine-tuning cannot save any computation. For parameter-shared tuning, W_{sub_P} can be written as $W_{base}+\delta W_{sub\_P}$. Although δW_{sub_P} can be very sparse, computation cannot be saved and reused between base and sub-tasks since activations vary between them. An example in Fig. 2 shows the density change in delta activation between base task and sub-task. Even if weights differ only slightly (3 out of 16) between the sub-task and base task after parameter-shared tuning, activations vary rapidly (6 of 8) after only the first layer for the same input. This dense delta activation makes

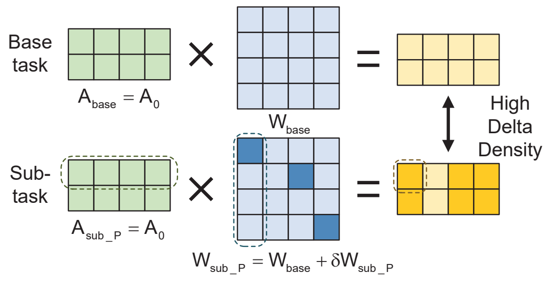

Figure 2: Delta activation density rises rapidly even in the first linear layer.

it difficult to share information between base task and sub-tasks and poses a challenge for computation saving in sub-tasks.

Recently, LeTS [11] proposed a new computation-friendly NLP transfer learning algorithm that can reduce 49.5% of computations for sub-tasks. However, LeTS changes the original BERT layer cascading structure by adding extra pooling and Bi-LSTM layers, limiting its application to classification tasks only without covering sequence-in-sequence-out tasks such as question-answering, NER, POS tagging, summarization, etc. Moreover, the LeTS network structure requires the storage of additional pre-trained intermediate activations during inference, which increases activation memory overhead.

In this paper, we propose *TaskFusion*, which to our knowledge is the first software-hardware co-optimized architecture design for efficient multi-task NLP. The main idea of *TaskFusion* is to boost data sharing between tasks for both weights and activations. First, we design an efficient transfer learning algorithm that enables the model to automatically learn data redundancies between the base task and sub-tasks. The training process maximizes the sparsity in delta weights and delta activations while retaining accuracy. *TaskFusion* can reduce both storage and computation in multi-task scenarios. Moreover, our algorithm can be combined with previous single-task acceleration methods to obtain more improvements. To unleash the full potential of our algorithm, we design a dedicated heterogeneous architecture, which consists of a dense core, a sparse core and an attention core. With a sparse core, the heterogeneous architecture can more efficiently accelerate sparse matrix-matrix multiplication (spMM) in sub-tasks. In addition, we propose a novel multi-task scheduling scheme to fully utilize the proposed architecture's resources and memory bandwidth, which further speeds up multi-task processing.

In summary, our work makes the following contributions:

- An efficient transfer learning algorithm that boosts weight and activation sharing between base task and sub-tasks and reduces both storage and computation when executing sub-tasks in multi-task processing.
- Recognizing that *TaskFusion* can be combined with previous single-task transformer accelerators, we present a novel heterogeneous architecture by adding a dedicated sparse computation core.
- A multi-task scheduling scheme that further improves the latency and efficiency through exploiting hardware utilization and memory bandwidth.
- Experiments and evaluations on the *TaskFusion* algorithm, architecture, and scheduling from accuracy, latency, and energy efficiency perspectives, covering more than 10 NLP tasks and 2 different pre-trained models. *TaskFusion* algorithm can save more than 98% storage and 73% computation per sub-task by using sparse delta weight and activation. *TaskFusion* architecture can achieve up to 3.75x speed-up and 2.86× energy efficiency compared to previous state-of-the-art single task accelerators under multi-task scenarios.

2 BACKGROUND

2.1 Pre-trained Model and Task-specific Fine-tuning Scheme

Task-specific fine-tuning has become a standard paradigm and demonstrated remarkable performance in various applications from vision [9, 17] to natural language processing [3, 7, 24, 34]. Pre-training is usually an unsupervised learning procedure performed on very large datasets. For instance, RoBERTa [24] uses five English-language corpora of varying sizes and domains, totaling over 160GB of uncompressed text. The pre-trained model (PTM) provides a good common initialization and basic token information for supervised learning of downstream tasks. The fine-tuning then trains task-specific models on downstream tasks (sentiment analysis, named entity recognition, etc.) by simply fine-tuning all pre-trained parameters as Fig. 3 shows. Suppose each sub-task $\tau \in \mathcal{T}$ has an associated dataset $\mathcal{D}_\tau = \{x_\tau^{(n)}, y_\tau^{(n)}\}_{n=1}^{N}$. For all tasks, task-specific fine-tuning aims to produce a set of model parameters w_τ to optimize the following constrained optimization problem:

$$\min_{w_\tau} \frac{1}{N} \sum_{i=1}^{N} C(f_\tau(x_\tau^{(n)}; w_\tau), y_\tau^{(n)}) + \lambda \mathcal{R}(\cdot) \tag{4}$$

where f_τ is a neural network function, $C(\cdot)$ is a cost function (e.g., cross-entropy) and $\mathcal{R}(\cdot)$ is an optional regularization function with hyperparameter λ.

2.2 Parameter-shared Tuning

Although task-specific fine tuning approach achieves good performance, the updating of all parameters in large-scale PLMs and storing all fine-tuned models still exhibits prohibitive adaptation costs.

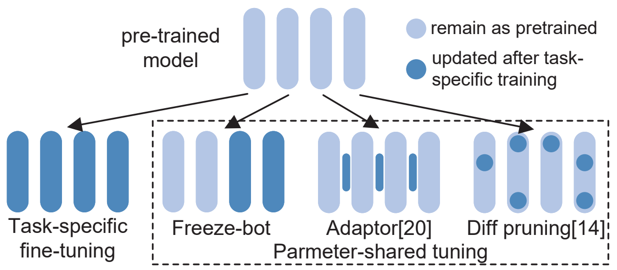

Figure 3: Pre-trained model and different weight tuning methods.

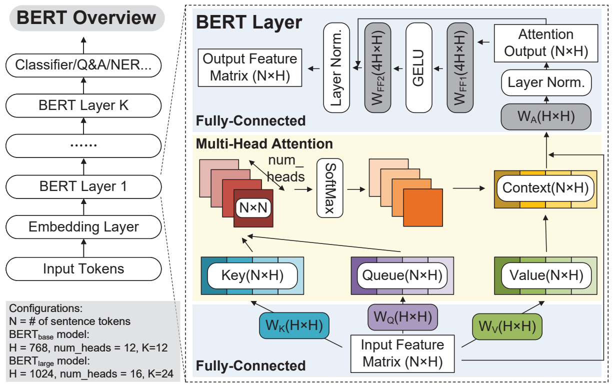

Figure 4: BERT structure overview.

To reduce the memory consumption, parameter-shared tuning is proposed to only update part of parameters in for downstream tasks. In traditional transfer learning schemes [30], parameter sharing is realized by only retraining the last several layers and freezing other layers. We refer to this method as freeze-bot in Fig. 3. However, recent studies [11, 13] show that freeze-bot leads to intolerably more than 10% accuracy drop. Adaptor [19] inserts parameter-shared sublayers between pre-trained model layers and only updates these added layers as depicted in Fig 3, which adds 3.6% extra parameters per task with no accuracy loss. Diff pruning [13] (Fig. 3) uses pre-trained weights plus sparse delta weights as the new task-specific weights. Experiments show that it can reduce the amount of extra parameters to around 0.5% per task with negligible accuracy loss.

2.3 Transformers and Computation Breakdown

Transformer structures have achieved state-of-the-art performance in various NLP tasks. BERT is one of the most widely adopted transformer-based structures. Fig. 4 shows the structure of BERT neural network. There are two main computation types in BERT layers: fully connected layer (linear layer) and multi-head self-attention. Previous transformer accelerators [14, 15, 23, 33, 48] mainly seek to accelerate the multi-head self-attention part since they consume most of the time in a CPU or GPU when the number of tokens is large. Fig. 5 shows the number of floating point operations (FLOPs) breakdown versus number of input tokens. For short input sequences such as less than 256 tokens, which are often observed in edge scenarios, linear layers occupy over 90% of the total operation counts. As input sequence token number N increases, the self-attention part becomes more significant since self-attention computation complexity is $O(N^2H)$ while linear layer complexity is $O(NH^2)$. Therefore, accelerating fully-connected layers is as significant as accelerating self-attention for system efficiency under edge scenarios, especially when input sentences are not long.

3 TASKFUSION ALGORITHM

3.1 Overview

Challenges mentioned previously show that computation overhead arises from the non-shareable weights and activations between the base task and sub-tasks. If both the weights and activations were shared, we could save computation by processing only sub-task specific delta parameters and features that deviate from the base task. Based on this observation, we propose *TaskFusion* algorithm, which is an efficient transfer learning algorithm that combines weight sharing or freezing, learning-based delta weight pruning, and learning-based delta activation pruning.

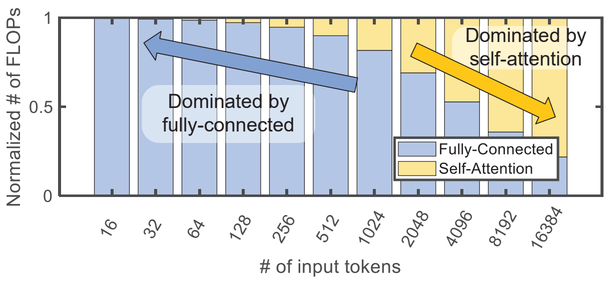

Figure 5: BERT computation breakdown versus number of input tokens.

Fig. 6 shows the network structure of BERT in *TaskFusion* algorithm, which is the same for base task and sub-tasks. We split the sub-task layers into three parts: totally shared layers (N_{TS}), partially shared layers (N_{PS}), and non-shareable layers (N_{NS}). In totally shared layers, weights are kept the same as the base task (pre-trained model). To describe the algorithm without loss of generality, we use the same linear layer L example from Fig. 1 (d). Since the input of a sub-task and the base task is the same (e.g., a sentence), the activation in the totally shared layers can be directly reused from the base task to sub-tasks without any computation as in Eq.5:

Totally shared layer L of a *TaskFusion* sub-task where $\mathrm{L} < \mathrm{N_{TS}}$:

$$\mathrm{A^{L+1}_{sub\_T}} = \mathrm{A^{L}_{base}} \times \mathrm{W^{L}_{base}} = \mathrm{A^{L+1}_{base}} \tag{5}$$

In partially shared layers, a sub-task weight/activation matrix equals the base task weight/activation matrix plus a sparse delta weight/activation matrix. First, layer L of the base task is processed,

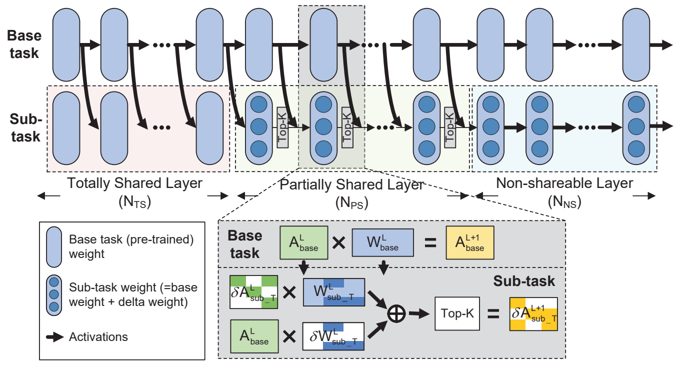
Base task
Sub-task
Top-K
Totally Shared Layer (N_TS)
Partially Shared Layer (N_PS)
Non-shareable Layer (N_NS)
Base task (pre-trained) weight
Sub-task weight (=base weight + delta weight)
Activations
Base task
A^L_base × W^L_base = A^{L+1}_base
Sub-task
δA^L_sub_T × W^L_sub_T
A^L_base × δW^L_sub_T
⊕ → Top-K = δA^{L+1}_sub_T

Figure 6: *TaskFusion* algorithm overview: we split BERT layers into 3 parts: totally shared layers, partially shared layers and non-shareable layers.

and then the activation during sub-task processing can be written as $A^L_{sub\_T} = A^L_{base} + \delta A^L_{sub\_T}$, where T stands for *TaskFusion*. δW_{sub_T} and δA_{sub_T} are the task-specific delta weight and delta activation matrices, respectively. The partially shared layer processing can be written as Eq.6 (also refer to Fig. 1 (d)):

Partially shared layer L of a *TaskFusion* sub-task where $N_{TS} \leq L < N_{TS} + N_{PS}$:

$$\begin{aligned} A^{L+1}_{sub\_T} &= A^L_{sub\_T} \times W^L_{sub\_T} \\ &= (A^L_{base} + \delta A^L_{sub\_T}) \times (W^L_{base} + \delta W^L_{sub\_T}) \\ &= A^L_{base} \times W^L_{base} + \delta A^L_{sub\_T} \times (W^L_{base} + \delta W^L_{sub\_T}) \\ &\quad + A^L_{base} \times \delta W^L_{sub\_T} \\ &= A^{L+1}_{base} + \delta A^L_{sub\_T} \times W^L_{sub\_T} + A^L_{base} \times \delta W^L_{sub\_T} \end{aligned} \tag{6}$$

Since δW_{sub_T} and δA_{sub_T} are sparse matrices, it is no longer necessary to perform a new dense matrix-matrix multiplication ($A^L_{sub\_T} \times W^L_{sub\_T}$). Instead, the pre-computed base task activation A^{L+1}_{base} can be reused in sub-task computation as shown in Eq. 6. By this sharing, two sparse-dense matrix-matrix multiplications ($\delta A^L_{sub\_T} \times W^L_{sub\_T} + A^L_{base} \times \delta W^L_{sub\_T}$) replace the dense matrix-matrix multiplication. When δA_{sub_T} and δW_{sub_T} are very sparse, this replacement can speed up sub-tasks and also reduce energy consumption. In order to make the output *delta* activation more sparse, we propose to select top-K largest absolute deltas and force the other activation deltas to zero before passing it to next layer:

$$\delta A^{L+1}_{sub\_T} = \text{top}_K(\delta A^L_{sub\_T} \times W^L_{sub\_T} + A^L_{base} \times \delta W^L_{sub\_T})$$

In non-shareable layers, activations are not shared but the delta weight matrix δW_{sub_T} is trained to be sparse via parameter-shared tuning to lower memory storage requirements. We do not constrain activations and therefore no computation savings are achieved in non-shareable layers. The computation of non-shareable layers can be written as the following:

Non-shareable layer L of a *TaskFusion* sub-task where $L \geq N_{TS} + N_{PS}$:

$$\begin{aligned} A^{L+1}_{sub\_T} &= A^L_{sub\_T} \times (W^L_{base} + \delta W^L_{sub\_T}) \\ &= A^L_{sub\_T} \times W^L_{sub\_T} \end{aligned} \tag{7}$$

In summary, memory savings are achieved in all three types of layers whereas computation savings are attained in totally shared layers and partially shared layers. To maximize the speed-up and energy saving, sparsifying δW_{sub_T} and δA_{sub_T} as much as possible remains a challenge. In the following sections, we introduce our learning-based delta weight and activation pruning methods to increase the sparsity.

3.2 Learning Sparser Delta Weights and Delta Activations

Diff pruning [13] uses regularization on delta weights between the pre-trained model and fine-tuning model to sparsify the delta weight matrix. Similarly, we apply regularization to *both delta weights and delta activations* between the base task and sub-tasks. Here, we change the notation for simplicity, only considering partially shared layers. Suppose the base task weight is w_p and activation is a_p, the sub-task weight is w_τ and activation is a_τ. We define the delta weight and delta activation as:

$$\delta w_\tau = w_\tau - w_p, \quad \delta a_\tau = a_\tau - a_p$$

In order to make δw_τ and δa_τ sparse, we modify the optimization loss function from Eq.4 to:

$$\min_{w_\tau} \frac{1}{N} \sum_{i=1}^{N} C(f_\tau(x_\tau^{(n)}; w_\tau), y_\tau^{(n)}) + \mathcal{R}_w(\delta w_\tau) + \mathcal{R}_a(\delta a_\tau) \tag{8}$$

Since diff pruning has proven the effectiveness of ℓ_0 regularization on sparsifying δw_τ, we use the same method to learn the sparse δw_τ by using:

$$\mathcal{R}_w(\delta w_\tau) = \lambda_w \sum_{l=0}^{L} ||\delta w_\tau^l||_0 = \lambda_w \sum_{l=0}^{L} \sum_{i=1}^{d} \mathbb{1}\{\delta w_{\tau,i}^l \neq 0\} \tag{9}$$

where L equals the total number of layers and λ_w is the regulation coefficient. Since Eq. 9 is non-differentiable, we also follow the approach in [25] and [13] for gradient-based learning using a relaxed mask vector. Once a binary mask is learned, we multiply this mask with dense delta weight matrix to make δw_τ sparse.

Interestingly, the same approach does not extend well to sparse delta activation learning. Since delta activation δa_τ also depends on the variable input x_τ, it is hard to learn a fixed deterministic binary mask applicable to all inputs. ℓ_1 regularization (also known as *Lasso*) [46] has been proven effective for regularizing parameters to generate a Laplacian-like distribution, increasing the amount of small values. Therefore, we use ℓ_1 regularization on delta activations during training to shape the distribution of δa_τ to have a higher probability of small values. The proposed regularization function of δa_τ is written as:

$$\mathcal{R}_a(\delta a_\tau) = \sum_{l=0}^{L} \lambda_a^l ||\delta a_\tau^l||_1 = \sum_{l=0}^{L} \lambda_a^l \sum_{i=1}^{d} |\delta a_{\tau,i}^l| \tag{10}$$

where λ_a^l is the layer-wise regularization hyperparameter. We set $\lambda_a^l (> 0)$ only for partially shared layers. For the other layers, we set $\lambda_a^l = 0$. As a result, absolute values of δa_τ can be reduced for partially shared layers, but they will not decrease to zero. Therefore, we only select top-K largest absolute values of δa_τ during inference. Sparsity of δa_τ is controlled by choosing the K value.

Fig. 7 shows the δa_τ distribution of BERT_{base} model 4th and 5th layer on sentiment analysis (SST-2) task with (a) no regularization, (b) ℓ_1 regularization, and (c) top-K selection after (b) (top 20% selected). Note that by using ℓ_1 regularization, delta activation distribution is significantly narrowed from Gaussian-like to Laplacian-like. The overall absolute value of delta activation also decreases. After top-K selection, small values are set to zero and only K largest values remain. This produces high sparsity for δa_τ and enables sparse matrix multiplications to replace dense multiplications.

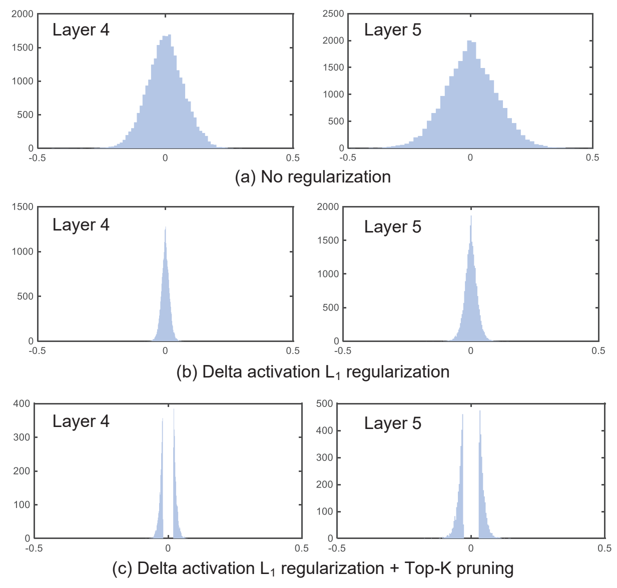

Figure 7: Delta activation distribution of BERT$_{base}$ 4th and 5th layer for the sentiment analysis (SST-2) task. x axis: delta activation value. y axis: density.

3.3 Combine with Previous Single-task Acceleration

Many efficient single-task transformer acceleration methods have been proposed, such as approximate attention [14, 15], quantization [51], token pruning [48], attention map pruning [23, 45], low-rank approximation [33]. Our algorithm is orthogonal to them and can combine with them to obtain more gains for multi-task processing. In this work, we choose to augment *TaskFusion* with token pruning (SpAtten) [48], as an example to demonstrate that adaptability. SpAtten prunes less important tokens in a cascaded way to reduce the activation matrix dimension and speed up. When combining Spatten, the base task computation is identical to the original *TaskFusion* without using token pruning. For sub-tasks, the token pruning is implemented in all layers by pruning small attention score tokens. Fig. 8 compares activation patterns in partially shared layers with and without combining SpAtten. Task-specific fine-tuning does not enable any activation sharing while *TaskFusion* does, making delta activation sparse in the partially shared layers. In SpAtten, token "ISCA" and "is" are identified as unimportant tokens. Therefore, only "fun" related activation is left and passed to the next layer. By adding *TaskFusion*, base task and sub-tasks partially share "fun" activations, with the tokens "ISCA" and "is" still pruned out. This combination makes the delta activation smaller and also sparser. Experiments show only negligible accuracy loss with this combination (Sec. 5).

4 TASKFUSION HARDWARE ARCHITECTURE

4.1 Overview

Existing transformer accelerators cannot efficiently support our *TaskFusion* algorithm, mainly due to four challenges:

- Handling irregular sparse matrix operations

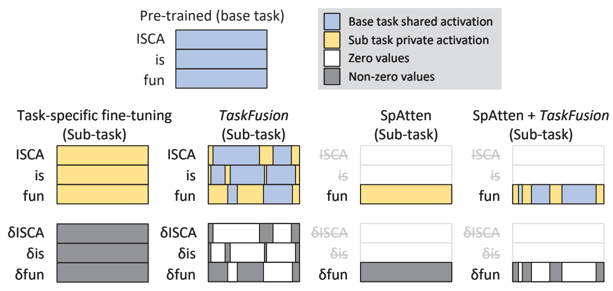
Pre-trained (base task)
ISCA
is
fun
Base task shared activation
Sub task private activation
Zero values
Non-zero values
Task-specific fine-tuning (Sub-task)
TaskFusion (Sub-task)
SpAtten (Sub-task)
SpAtten + TaskFusion (Sub-task)
ISCA
is
fun
δISCA
δis
δfun

Figure 8: Partially shared layer activation pattern comparison between task-specific fine-tuning, *TaskFusion*, SpAtten, and SpAtten + *TaskFusion*.

- Memory architectural support for sparse delta matrices
- Multi-task scheduling for data reuse and reduced off-chip memory accesses
- Joint acceleration of self-attention layers and fully-connected layers.

To tackle these challenges, *TaskFusion* proposes a heterogeneous architecture that consists of a dense core, a sparse core, and an attention core to accelerate different computation patterns. We design a new on-chip memory architecture supporting delta weights and delta activations. We also propose a novel multi-task scheduler to efficiently orchestrate tasks.

4.2 Heterogeneous Architecture

Fig. 9 shows the proposed *TaskFusion* architecture. For the base task and non-shareable layers of sub-tasks, weights and activations are dense. Hence for those, the dense core is used to compute matrix multiplications for Queue (Q), Key (K), Value(V) and feed-forward networks (FFN). For the dense core, we use an output stationary TPU-like systolic array architecture[20] shown in Fig. 9 ❶. Each PE receives weights from the left and activations from the top, computes multiplication and addition (MAC) operations, and stores the partial sum to the PE-local register. Then the weights and activations are passed in a systolic manner to neighbor PEs through each PE's right and bottom connections. After a chunk of computation is finished, partial sums are sent to the accumulation memory for accumulation with previous partial sums. Finally, the outputs are sent to non-linear units like GELU and layer normalization, or directly sent to the attention core for attention operations. The base task activations are stored in the base task activation memory and attention core's K, Q, V memory.

After K, Q and V being computed by the dense core , attention computation starts in the attention core shown in Fig. 9 ❷. The attention core computes QK multiplication, SoftMax, attention score (S) and V multiplication, etc., to obtain the attention output (A). There are two reasons for not reusing computation units between attention core and dense core: 1) the computation pattern is highly different between fully-connected matrix computation and attention matrix computation, where the latter requires matrix transposition, SoftMax, normalization, etc. and 2) using two cores enables pipelining between Q generation and QK multiplication, which reduces overall latency. In our design, Key memory and Value

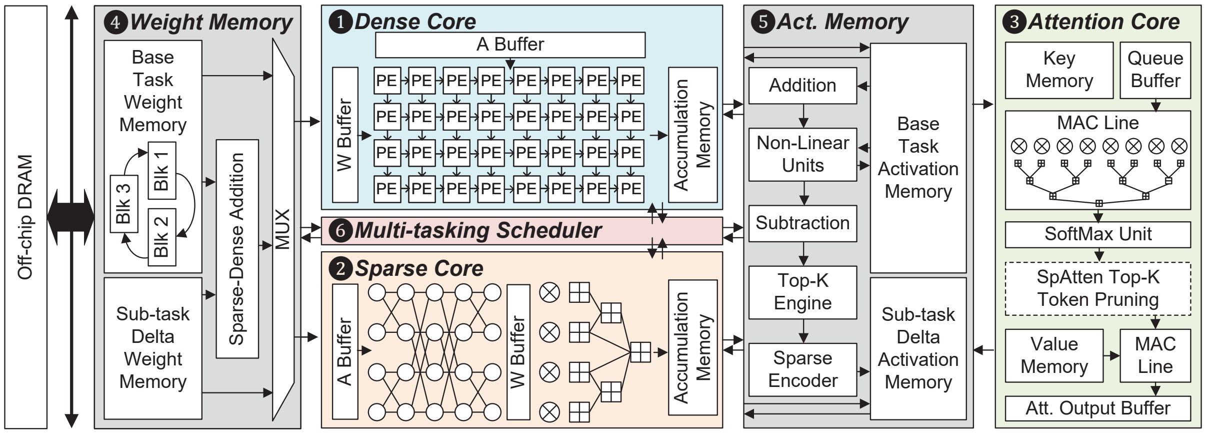

Figure 9: *TaskFusion* architecture.

memory are both 192KB for storing up to 128 token features. Moreover, with potential combination of *TaskFusion* and other single-task acceleration methods, the attention core may be replaced. Still taking SpAtten as an example, we add its top-K pruning block after the SoftMax unit (dashed box in Fig. 9 ❸) for on-the-fly token pruning. Attention core is used for both the base task and sub-tasks. The attention result of the base task is directly stored in the base task activation memory. For sub-tasks, base task attention results are first subtracted from sub-task results, then the delta results are sparsified by choosing the top-K largest and stored in a bit-map encoded format in the delta activation memory. These hardware supports are implemented in the activation memory block (Fig. 9 ❺).

During the execution of partially shared layers in sub-tasks, we use the sparse core to handle irregular sparsity and accelerate the two sparse matrix multiplications in Eq.6 since the non-zero elements in δW_{sub_T} and δa_{sub_T} are chosen dynamically based on their magnitudes. There have been many sparse DNN accelerators [12, 16, 18, 31, 32, 40, 43, 44, 53, 54] in recent years to accelerate sparse matrix multiplications. We choose SIGMA-like [32] architecture (Fig. 9 ❸) since it has the advantage of handling irregular matrix sizes (when combining Spatten) as well as irregular matrix sparsities, thanks to SIGMA's flexible and configurable distribution network and reduction network. In totally shared layers and non-shareable layers, the sparse core is power-gated to save power consumption since no sparse computation is needed. In partially shared layers, the sparse core first computes $\delta A_{sub\_T} \times W_{sub\_T}$. Base task and sub-task delta weights are read from the base task weight memory and sub-task delta weight memory, respectively, and added together using sparse-dense addition units, then passed to the sparse core's W buffer. Delta activations are read from the sub-task delta activation memory and stored in the A buffer. After sparse-dense matrix multiplication, results are stored in the accumulation memory, waiting for the second sparse matrix multiplication $A_{base} \times \delta W_{sub\_T}$ that reads δW_{sub_T} from the sub-task delta weight memory and A_{base} from the base task activation memory. The final results are first added to the base task activation, then go through non-linear units. Delta activations are obtained by subtracting the post-GELU/LayerNorm base task activations from the resulted activations. Here, we only store the pre-GELU/LayerNorm base task activations and LayerNorm coefficients to decrease the on-chip memory overhead as well as off-chip memory accesses. Before subtraction, we on-the-fly redo GELU or LayerNorm (use stored coefficients) on the stored base task activations. Finally, we use the Top-K engine to sparsify delta activations, and store the encoded sparse delta matrix in the sub-task delta activation memory.

For a prototype accelerator, we use 16 bit floating point multipliers and adders in MAC PEs without losing accuracy. We instantiate 256 multipliers in the dense core and sparse core, and 128 multipliers in the attention core for pipelined attention computation, totaling 640 multipliers. However, *TaskFusion* is salable with a different number of multipliers. For sensitivity study, we instantiated a small version with total 160 multipliers and a medium version with total 320 multipliers. Detailed evaluation results are shown in Sec.5.

4.3 Memory architecture

First, we separate the base task memory and sub-task delta memory to increase the memory bandwidth. This enables parallel execution of the base task and sub-task. The weight memory system contains the base task weight memory, sub-task delta weight memory and sparse-dense addition unit. For the base task weight memory, we use a circular-buffer-like architecture, which contains three memory blocks as Fig. 9 ❹ shows. Each memory block is 288KB and this relatively large size is determined to decrease the off-chip memory access. Each memory block rotates its role from base task weight supply, sub-task weight supply, and weight pre-loading from off-chip. With our scheduling method, this design can reduce the off-chip pre-trained (base task) model weight access to only once to perform all tasks in our experiments.

For the sub-task delta weight memory, we use a bit-map (binary mask) to encode the delta weight, which aligns with the SIGMA design. The sub-task delta weight memory size is set to 256KB. Since we need to calculate $W^L_{base} + \delta W^L_{sub\_T}$ in Eq. 6, we design a sparse-dense addition unit in the weight memory system. First, the

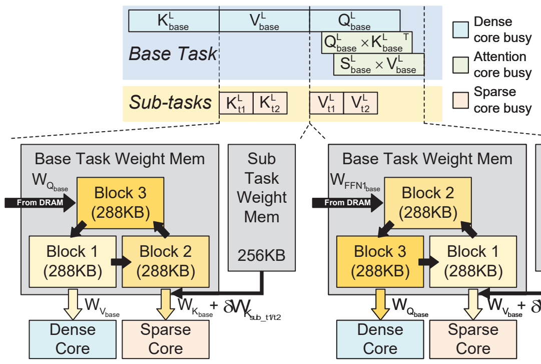

Figure 10: *TaskFusion* weight memory architecture and control flow.

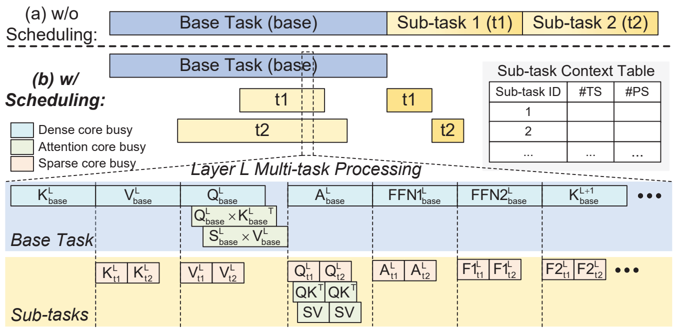

Figure 11: *TaskFusion* scheduling scheme: (a) Naive scheduling, where sub-tasks are executed one by one after the base task. (b) Proposed scheduling, which pipelines layer execution of the base task and sub-tasks to reduce latency and off-chip memory access.

non-zero delta weight positions are identified and stored while computing $A^L_{base} \times \delta W_{sub\_T}$. The sparse-dense addition unit then uses the stored non-zero delta weight positions to add delta weights with base weights. The sparse-dense addition unit contains 32 adders.

The detailed weight memory micro-architecture and control flow are shown in Fig. 10. The base task and sub-tasks are scheduled in a pipelined manner following the schedule shown in Sec. 4.4. When the dense core is computing V (Fig. 10 left), V weights ($W_{V_{base}}$) are read from block1, and the sparse core generates sub-task K. The base K weights ($W_{K_{base}}$) are read from block2, which are added to sub-task (t1 or t2) delta K weights ($\delta W_{K_{sub\_t1/t2}}$) and sent to the sparse core. Meanwhile, block3 receives the next layer's weight ($W_{Q_{base}}$) from DRAM. For the next stage (Fig. 10 right), the sparse core can reuse $W_{V_{base}}$ from block1. Since $W_{K_{base}}$ is no longer useful, block2 is updated with next layer's weights ($W_{FFN1_{base}}$). This rotation eliminates on-chip weight access conflicts and also minimizes off-chip weight accesses from DRAM.

The activation memory system (Fig. 9 ❺) consists of base task activation memory, sub-task delta activation memory, addition/subtraction units, GELU and layer normalization units, top-K engine, and a sparse data encoder. We use a bit-map to encode the sparse sub-task delta activation. Each addition/subtraction unit contains 32 adders/subtractors. We use top-K engine to determine the K-th largest value and generate a bit-map by comparing elements with a pre-defined threshold. The resulted sparse activation is encoded by the sparse encoder to be stored in the sub-task delta activation memory. In our design, we set base task activation memory to be 576KB and sub-task delta activation memory to be 256KB. If there are many sub-tasks executing at the same time, off-chip memory access is needed.

4.4 Task Scheduling

In order to maximally utilize computation resources and reduce off-chip memory access in multi-task processing, we propose a *TaskFusion* scheduling scheme depicted in Fig. 11, where we compare the processing flow with and without *TaskFusion* scheduling. A conventional processing flow performs sub-tasks one-by-one after the base task processing, which is costly because 1) all base task activations need to be stored off-chip for later sub-task processing, 2) pre-trained weights and base task activations are reloaded from off-chip every time a new sub-task is performed and 3) the computation cores are under-utilized since the dense core and sparse core are not used at the same time. Our proposed scheduling addresses those problems. First, sub-task information (e.g., number of totally and partially shared layers) is stored in a sub-task context table. When base task processing reaches one of the sub-tasks' partially shared layers, the related sub-task processing starts. Multiple sub-tasks can be scheduled to execute together as the example layer breakdown shown in Fig. 11. The attention computation is pipelined with a queue for both base task and sub-task based on the approach in [10]. Moreover, the sub-task processing is pipelined with the base-task processing for sharing the base task weights and activations which are necessary for *TaskFusion*. This pipelining can hide the latency of partially shared layers of sub-tasks behind base task processing. When the total latency of multiple sub-tasks exceeds that of the base task, the scheduler stalls base task processing to wait for sub-tasks to finish.

With our dedicated memory architecture and optimized scheduler, almost all coordination overhead between different computing and memory blocks can be eliminated. The complexity of *TaskFusion* scheduling is low. Hence it can be implemented using a lightweight low-power RISC processor such as Cortex-M0 with 32KB SRAM for the scheduler's instruction and data memory.

5 EVALUATION

5.1 Evaluation Setup

5.1.1 Workloads. Since BERT has shown great potential on various NLP applications, we use BERT as our backbone model and tested it with two configurations: $\text{BERT}_{\texttt{base}}$ and $\text{BERT}_{\texttt{large}}$. In our evaluation, we use 10 sub-tasks in total. Among them, 8 tasks are from the GLUE dataset [47], including linguistic acceptability (CoLA [49]), sentiment analysis (SST-2 [42]), natural language inferences (RTE [6], QNLI [35], MNLI [50]), sentence similarity detection (MRPC [8], QQP, and STS-B [4]). Since those are all sentence classification tasks, we add two non-classification tasks: question-answering (SQUAD [36]) and named entity recognition (NER [39]) to show

TaskFusion's generalizability. All tasks share the same $\mathtt{BERT}_{\mathtt{base}}$ / $\mathtt{BERT}_{\mathtt{large}}$ pre-trained model for the masked sentence and next-sentence prediction task [7] as the base task. We do not train all sub-tasks together since we lack a common training dataset for all 10 sub-tasks. Instead, we train each sub-task separately using each sub-task's dataset (based on the pre-trained model for the base task).

5.1.2 Hardware Settings. For latency estimation, we use an open-source systolic-array simulation tool (Scale-Sim [37]) for dense core cycle-accurate simulations. We use an open-source SIGMA simulator (STONNE [28]) for sparse core cycle-accurate simulations. For the attention core, we build our own cycle accurate model for both traditional attention computation and SpAtten. For energy simulation, we use our own dense core and attention core Verilog implementations. We use SIGMA's open-source RTL [32] for sparse core energy simulations. All Verilog designs are synthesized using Synopsys Design Compiler in TSMC 22nm ULL technology at the frequency of 1GHz and we use PrimePower with FSDB waveform to estimate the energy consumption. For memory energy consumption, we use ARM 22nm memory compilers to generate the memories we need. We use Cortex-M0 as a multi-task scheduler to control computing cores and memory blocks. For end-to-end processing latency, we build a system-level cycle-accurate model that considers the proposed scheduling and Cortex-M0 control latency. For system energy consumption, we add different cores' computation energy, memory access energy, and Cortex-M0 control energy together to estimate the overall energy of *TaskFusion*.

5.2 Algorithm Evaluation

We mainly compare 4 different algorithms. The baseline algorithm is the conventional task-specific fine-tuning scheme, without sharing between the base task and sub-tasks. The SpAtten algorithm stands for the token-pruning method proposed in [48]. It employs token pruning to reduce the number of tokens during inference, but it also does not enable sharing between the base task and sub-tasks. The *TaskFusion* algorithm is our method with delta weight and delta activation pruning, which enables parameter and activation dual sharing between the base task and sub-tasks. The SpAtten + *TaskFusion* algorithm is the combination of our method with token pruning as discussed in Sec. 3.

For each sub-task in *TaskFusion*, we set the number of totally shared layers, the number of partially shared layers, delta weight sparsity, and delta activation sparsity as extra hyperparameters. For SpAtten and SpAtten + *TaskFusion*, we set the first 3 layers to keep 100% tokens and set a hyperparameter for the final layer's token keep rate. The token keep rates for remain layers linearly decrease from 100% to the token keep rate of the final layer.

We use a three-step training scheme adapted from diff pruning [13] with the addition of delta activation pruning. First, ℓ_0 regularization on delta weights and ℓ_1 regularization on delta activations are added to the loss function. Each input batch (a batch of sentences) goes through the pre-trained model and we store all intermediate activations as base activations. Then during the forward-propagation of the sub-task model training, we use those base activations as the reference to obtain sub-task delta activations. From those, we only keep partial delta activations with the largest absolute values based on the pre-set delta activation sparsity (10% - 20%) in partially shared layers. In non-shareable layers, l_1 regularization coefficient (λ_a^l, refer to Eq.10) is set to 0 and no delta activation pruning is attempted. Training a few epochs with those configurations provides the first stage training results. In the next stage training, we only keep a small portion ($\leq 2\%$) of the largest delta weights in partially shared layers and non-shareable layers to attain the pre-set delta weight sparsity. In the final stage training, we only update the non-zero delta weights at fixed positions with the same delta activation pruning strategy as in the first step. We enumerate different combinations of hyperparameters until we find a good combination with negligible accuracy drop.

Fig. 12 shows the accuracy and computation comparison of 2 model structures ($\mathtt{BERT}_{\mathtt{base}}$ and $\mathtt{BERT}_{\mathtt{large}}$) on 10 sub-tasks. For $\mathtt{BERT}_{\mathtt{base}}$ model, *TaskFusion* and SpAtten has negligible accuracy loss (around 0.5% - 0.8%) on average (AVG). SpAtten + *TaskFusion* has larger accuracy loss (around 1.7%) but saves more computation, enabling a reasonable trade-off between accuracy and complexity. The same conclusion can be drawn for $\mathtt{BERT}_{\mathtt{large}}$. *TaskFusion* can save 65.2% / 62.4% FLOPs on average for $\mathtt{BERT}_{\mathtt{base}}$ / $\mathtt{BERT}_{\mathtt{large}}$ while SpAtten+*TaskFusion* increases FLOPs saving to 73.1% / 69.1% for $\mathtt{BERT}_{\mathtt{base}}$ / $\mathtt{BERT}_{\mathtt{large}}$ compared to the baseline.

We conducted additional experiments to show the benefits of ℓ_1 regularization on delta activations. Fig. 13 shows the accuracy of sub-task SST-2 and QNLI in *TaskFusion* (no SpAtten) with varying layer-wise ℓ_1 regularization coefficients λ_a^l (refer to Eq.10) when the target delta activation sparsity is set to 20%. $\lambda_a = 0$ means there is no ℓ_1 penalty on delta activations. $\mathtt{BERT}_{\mathtt{base}}$ model is used in this experiment. The red horizontal dotted line is the baseline task-specific fine-turning accuracy and the gray dotted line is drawn at 1% accuracy drop. Results show that removing ℓ_1 regularization causes large accuracy drop (> 2%). Using a small ℓ_1 regularization coefficient improves accuracy. However, when λ_a is too large, accuracy decreases. In our training for different sub-tasks, we set λ_a as a hyperparameter and choose the value that obtains the best accuracy for each sub-task.

In order to show the generality of our methods, we also test *TaskFusion* algorithm on another transformer model RoBERTa [24] with different sub-tasks and associated datasets. The results are shown in Table 1. *TaskFusion* achieves 65.8% number of FLOPs reduction on average for RoBERTa tasks with less than 1% average accuracy loss. This shows that *TaskFusion* can be generalized to multiple transformer models including BERT and RoBERTa.

Table 1: Accuracy and computation comparison between baseline and *TaskFusion* algorithm on $\mathtt{RoBERTa}_{\mathtt{base}}$

	MRPC	STS-B	RTE	SST-2	QNLI	MNLI	AVG
Baseline Acc.	92.2	90.8	80.1	94.2	91.9	87.8	89.5
TaskFusion Acc.	91.3	89.3	78.5	94.0	91.7	86.8	88.6
#FLOPS reduction	54%	59%	64%	72%	69%	76%	65.8%

5.3 Comparison with Baseline Design

5.3.1 Baseline Description. Hardware evaluation of this work first considers the **one-base-one-sub-task** scenario, where there is only one sub-task and we calculate latency and energy efficiency for only that sub-task. We compare four different architectures:

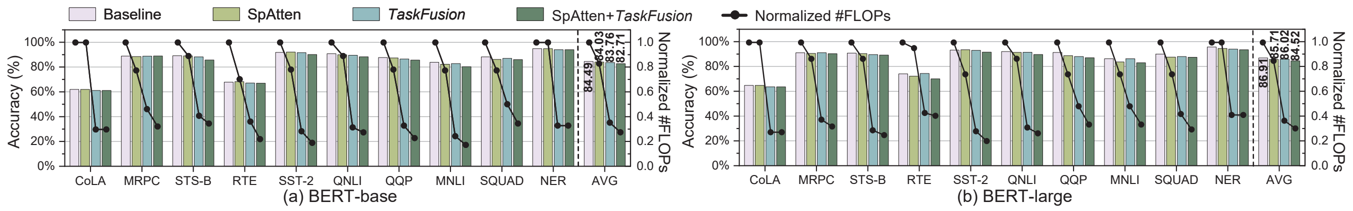

Figure 12: Accuracy (bars) and computation (number of FLOPs, lines) comparison between baseline, SpAtten, *TaskFusion* and SpAtten + *TaskFusion* algorithms on 10 sub-tasks and 2 pre-trained models. The average accuracy and average number of FLOPs are shown in the AVG bars.

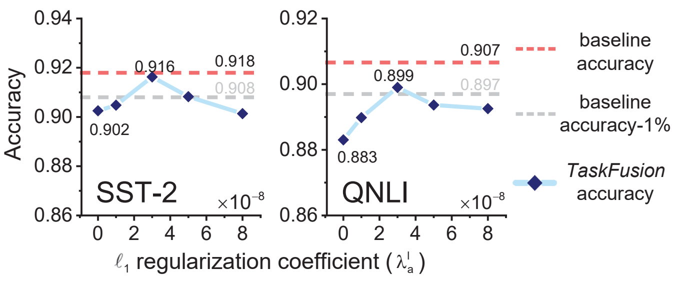

Figure 13: Accuracy comparison with different ℓ_1 regularization coefficient λ_a^l. $\lambda_a^l = 0$ means there is no ℓ_1 penalty on delta activations.

baseline, SpAtten, *TaskFusion*, and SpAtten + *TaskFusion*. The baseline architecture only contains dense and normal attention cores, without storage support for base task activations. Therefore there is no data sharing between the base task and sub-task. The sub-task needs to be calculated after the completion of base task. The SpAtten architecture enables on-the-fly token pruning by adding top-K pruning units to the attention core, which decreases the activation matrix (K, Q, V, attention, etc.) sizes and speeds up processing. However, SpAtten architecture cannot support weight and activation sharing between the base task and sub-tasks. *TaskFusion* is based on the architecture shown in Fig. 9, which contains dense, sparse, and attention cores as well as other computation and memory units to support data sharing between the base task and sub-tasks. SpAtten + *TaskFusion* architecture combines the two by adding the SpAtten top-K token pruning units to the attention core while keeping the support for data sharing.

5.3.2 Performance Comparison. Fig. 14 (a) and (b) show the latency comparison between the above four architecture settings for $\text{BERT}_{\text{base}}$ and $\text{BERT}_{\text{large}}$. The average sub-task speed up with *TaskFusion* is 2.85× for $\text{BERT}_{\text{base}}$ and 2.69× for $\text{BERT}_{\text{large}}$. The speed-up is mainly from layer-skipping in totally shared layers (contributes around 35%) and sparse computation in partially shared layers (contributes around 65%). Moreover, with SpAtten token pruning, the speed-up is around 3.55× for $\text{BERT}_{\text{base}}$ and 2.96× for $\text{BERT}_{\text{large}}$ compared to only using SpAtten, and 4.16× for $\text{BERT}_{\text{base}}$ and 3.35× for $\text{BERT}_{\text{large}}$ compared to the baseline architecture.

Fig. 14 (c) and (d) show the energy efficiency comparison for $\text{BERT}_{\text{base}}$ and $\text{BERT}_{\text{large}}$. The energy savings are from computation-skipping in totally shared layers, and sparse computation in partially shared layers. *TaskFusion* increases the energy efficiency by 1.55× − 2.43× (average 2.05×) for different tasks compared to the baseline for the $\text{BERT}_{\text{base}}$ case. For the $\text{BERT}_{\text{large}}$ case, the energy efficiency is increased by 1.69× − 2.77× (average 2.04×). By adding SpAtten, the energy efficiency is boosted to 2.79× − 2.94× compared to the baseline architecture. The energy efficiency gain is smaller than the speed-up rate since the sparse core consumes more power and there are other overheads such as top-K selections.

5.3.3 Area and Energy Breakdown. Fig. 15 shows the area and energy breakdown for the *TaskFusion* architecture. The energy breakdown is different for each sub-task, thus the numbers shown in Fig. 15 are the average numbers from all sub-tasks. Three major computation cores consume nearly 90% of total energy consumption. "Others" refers to the additional overhead incurred by top-K selection, addition/subtraction due to non-linear functions and multi-task scheduling.

5.3.4 Scheduling Overhead. As mentioned in Sec.4.4, the proposed memory micro-architecture design and pipelined layer execution eliminate most of the scheduling overhead. Therefore, a lightweight low-power RISC processor (e.g., Cortex-M0 with 32KB SRAM) is sufficient as the scheduler in *TaskFusion* design. For a detailed analysis, the scheduling scheme is implemented as C programs on Cortex-M0 that include the sub-task context table search, control-register write, controlling of different computation cores, etc. Behavioral simulations of the Cortex-M0 RTL (Verilog HDL) running the C programs show that the proposed scheduling only takes 0.09% of the end-to-end latency. Synthesis results show that the average power consumption of Cortex-M0 is only 2.4mW (0.3% of the total power) and the area overhead is only 0.06mm^2 (1.2%) that are negligible compared to the total system power and area.

5.4 Comparison with CPUs and GPUs

In this section, we compare *TaskFusion* with 2.4GHz Intel Xeon Gold CPU and NVIDIA V100 GPU. For a fair comparison, we follow a similar method as in DOTA [33] to scale up *TaskFusion* hardware resources to make it have the same peak throughput as NVIDIA V100 GPU (14 TFLOPS). We also scale the energy consumption up accordingly. Fig. 16 (left) shows the average normalized speed-up and energy efficiency of CPU, GPU, and *TaskFusion* on $\text{BERT}_{\text{base}}$ and $\text{BERT}_{\text{large}}$. In summary, *TaskFusion* achieves 141.4× / 14.2×

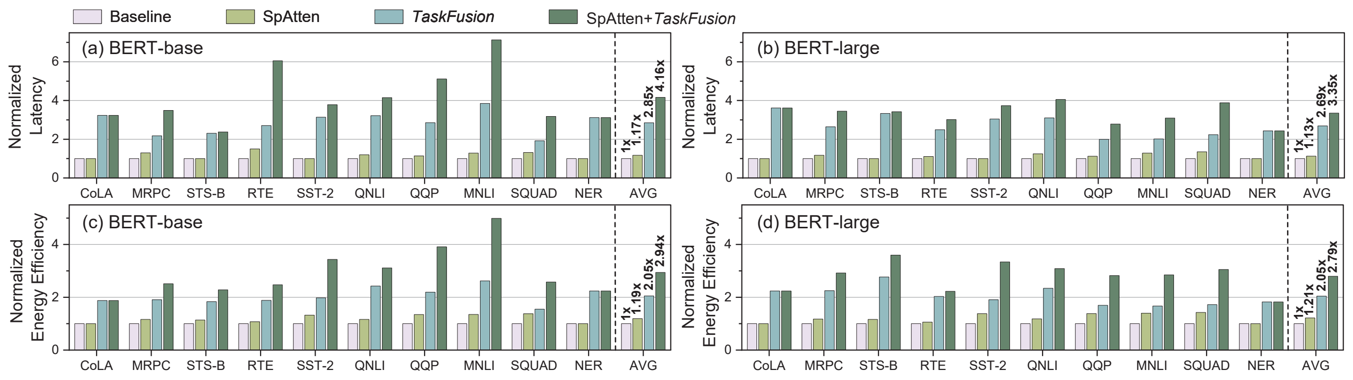

Figure 14: Normalized latency and energy efficiency comparison between baseline, SpAtten, *TaskFusion* and SpAtten + *TaskFusion* architectures on 10 sub-tasks and 2 backbone models. The average latency and energy efficiency are shown in the AVG bars.

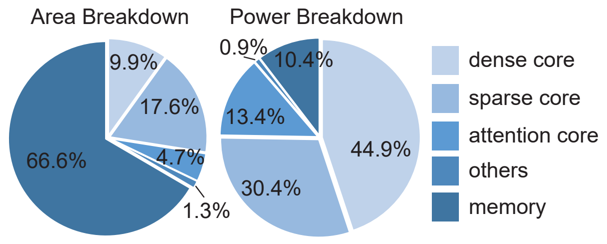

Figure 15: Energy and area breakdown of *TaskFusion* architecture.

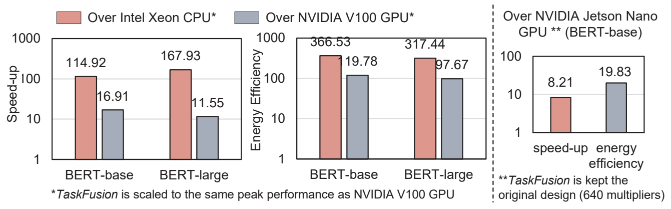

Figure 16: Speed-up and energy efficiency of *TaskFusion* over CPU and GPUs on end-to-end BERT inference.

speed-up and 341.9× / 108.7× higher energy efficiency over CPU / GPU on average for performing sub-tasks.

Since our design targets edge devices. We compare the original (without scaling) *TaskFusion* design (640 multipliers) with an edge GPU NVIDIA Jetson Nano. Fig. 16 (right) shows *TaskFusion* can achieve 8.21× speed-up and 19.83× energy efficiency comparing to the edge GPU for $\mathrm{BERT_{base}}$ end-to-end execution of sub-tasks.

5.5 Comparison with SOTA Accelerators

Table 2 shows the comparison between *TaskFusion* and state-of-the-art single-task transformer accelerators on end-to-end BERT inference. In order to make fair comparison, we follow the normalization method in [10] to make the total number of multipliers the same as *TaskFusion* (640 multipliers), and all designs use the same clock frequency of 1GHz. We implement SpAtten + *TaskFusion* for our architecture in this comparison and we use end-to-end $\mathrm{BERT_{base}}$ processing for all tasks. As the performance scales with the sentence length, we use normalized units for throughput (token/s) and efficiency (token/J) evaluation. The performance comparison for base task processing is different from that for sub-task processing. The base task performance in Table 2 of *TaskFusion* is evaluated by executing each given (sub) task as a base task without other sub-tasks that are executed using TaskFusion features. The sub-task performance in Table 2 is measured only for a sub-task executed using TaskFusion techniques with a main task when 1) there is only one sub-task without optimized scheduling, and 2) the sub-task is executed sequentially after completing the base task. This is equivalent to the performance of *TaskFusion* performing infinitely-many sub-tasks with optimized scheduling, where the base task latency can be completely amortized. Therefore, the base task and sub-task performances of *TaskFusion* in Table 2 can be understood as the lower bound and upper bound performances of multi-task NLP systems using *TaskFusion*. These performance bounds are illustrated in Fig.17, which shows the throughputs of *TaskFusion* with different numbers of sub-tasks. Since previous accelerators do not distinguish sub-tasks from the base task, the performance of executing sub-tasks in those accelerators is the same as that of the base task. *TaskFusion* has a lower base task throughput compared to prior works because we (optimistically) assume these prior works can fully utilize all 640 multipliers to maximize their performance. On the other hand, when *TaskFusion* executes the base task, only the dense core (256 multipliers) and attention core (128 multipliers) are utilized while the sparse core (256 multipliers) remains unused, as we assume fully dense execution of the base task. That leads to idling 40% of the multipliers. When executing sub-tasks, as shown in Table 2, *TaskFusion* achieves 1.48-2.43× speed-up and 1.62-3.77× higher energy efficiency than state-of-the-art single-task transformer accelerators.

5.6 Design Space Explorations

Table 3 shows the performance comparison between different designs of *TaskFusion*. We evaluate a small version totaling 160 multipliers and a medium version totaling 320 multipliers. Smaller versions with lower power consumption are tailored for resource-constraint platforms. The throughput of smaller versions show

Table 2: Comparison with state-of-the-art ASIC based transformer accelerators

	A^3[14]	SpAtten[48]	Sanger[26]	DOTA[33]	***TaskFusion***	
Technology	40nm	40nm	55nm	22nm	22nm	
Frequency	1GHz					
# of Multipliers	640					
Cross-task Sharing	No				Yes	
On-chip Memory [MB]	0.30	0.38	0.50	2.50	2.00	
Area [mm^2]	10.4	7.75	10.56	2.57	5.13	
Task Type	base task and sub-tasks have identical performance				base task	sub-task
Power [W]	1.337	1.165	0.880	0.942	0.635	**0.864**
Throughput [token/s]	3998.4	4588.4	4953.8	6566.3	2666.5	**9737.2**
Efficiency [token/J]	2988.7	3938.0	5626.9	6963.4	4195.8	**11261.6**

Table 3: Comparison with different size of TaskFusion prototype

	Small	Medium	Large
Technology	22nm		
Frequency	1GHz		
# of Multipliers	160	320	640
Memory [MB]	0.75	1.25	2.00
Area [mm^2]	1.82	3.29	5.13
Power [W]	0.275	0.488	0.864
Throughput [token/s]	2742.9	5179.4	9737.2
Efficiency [token/J]	9966.1	10624.1	11261.6

near-linear scaling while their efficiency stay close to that of the large version. It shows that *TaskFusion* architecture exhibits a reasonable scalability.

5.7 One-base-multiple-sub-tasks Scenario

We now evaluate a more practical **one-base-task-multiple-sub-tasks** scenario, where multiple sub-tasks are executed together with the same input. We apply our scheduling method to optimize the latency and energy efficiency. Our evaluation considers a system with 5 sub-tasks: MRPC, STS-B, QNLI, MNLI, RTE. Fig. 17 (a) shows the latency comparison with and without *TaskFusion* scheduling. The x-axis of Fig. 17 shows scenarios with different number of sub-tasks, where sub-tasks are added one by one. The base task and all sub-tasks receive the same input sequence, and we set the number of input tokens to be 46, which is the average token number in the above tasks' training dataset. When there is no scheduling, tasks are executed one by one as mentioned in Fig. 11 (a). *TaskFusion* and SpAtten + *TaskFusion* architectures reduce the latency of a single sub-task, thus reduce the total system latency accordingly. By adding the proposed scheduling, the system latency can be further reduced. With 1 base task and 5 sub-tasks, the SpAtten + *TaskFusion* architecture can reduce 60.3% of the total system latency (equals to 2.52× system speed-up). By adding scheduling, the total system latency reduction increases up to 67.7% (equals to 3.10× system speed-up).

Fig. 17 (b) shows the overall throughput of realistic multi-task NLP systems vs. the number of sub-tasks. When there is only the base task, *TaskFusion* throughput hits the lower bound. As the number of sub-tasks increases, *TaskFusion* throughput enhances and starts to surpass other state-of-the-art approaches, and eventually approaches the upper bound. Prior works do not distinguish sub-tasks from the base task, hence the throughput is the same regardless of the number of sub-tasks. The figure shows that with more than one sub-task, the throughput of *TaskFusion* is higher than SpAtten [48], and with more than two sub-tasks, *TaskFusion* outperforms DOTA [33].

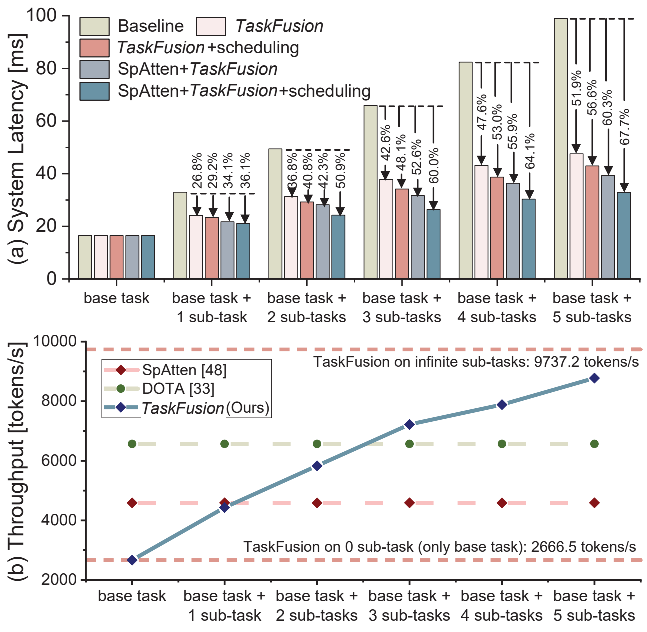

Figure 17: System latency and throughput comparison with different numbers of sub-tasks. The sub-task adding order is: MRPC, STS-B, QNLI, MNLI, and RTE.

6 RELATED WORKS

Hardware Support for Transformers. Many hardware accelerators have recently been proposed for accelerating transformer-like neural networks. A^3 [14] is the first work to apply approximation on SoftMax-based attention to speed-up the attention calculation. ELSA [15] also uses approximation computation to speed-up the transformer computation by using sign random projection. SpAtten [48] implements on-the-fly token pruning to make the attention matrix smaller, DOTA [33] uses low-rank linear transformation to detect and omit unimportant attention connections. Li, *et al* [23] adopt a gradient-based learning method to make the attention map sparse. To further exploit the attention sparsity, Energon [56] uses a low-precision network to predict the sparsity in attention map. To eliminate the unstructured sparsity overhead, Sanger [26] proposes pack-and-split modules to balance the computation. However, FABNet [10] points out that fully-connected layers occupy over 80% of operation counts for short input sequences. Therefore, accelerating the fully-connected like layers is also important. SpAtten [48] can reduce all matrix sizes (for both attention and fully-connected layers) and GOBO [51] uses quantization to accelerate all parts of transformer. FTRANS [22] and FABNet [10] use

FFT operations to replace traditional fully-connected operation. However, the works mentioned above are all for single task acceleration. To our knowledge, *TaskFusion* is the first work to accelerate multi-task NLP processing. Our work can also combine with those single task acceleration methods as mentioned in previous sections.

Multi-task Accelerators. There are few accelerators that aim at achieving multi-task processing. AI-MT [2] proposes a novel accelerator architecture that enables a cost-effective, high performance multi-neural network execution by fully utilizing the accelerator's computation resources and memory bandwidth. PREMA [5] uses a predictive multi-task scheduler to meet the latency demands of high priority tasks as well as maintaining high throughput. HDA [21] proposes a heterogeneous architecture that enables coarser-grained dataflow flexibility to increase the computation resource utilization. As opposed to previous multi-task architectures that seek to support different neural network structures, *TaskFusion* follows the task-specific fine-tuning scheme in NLP applications and uses one network structure (BERT, RoBERTa, GPT, etc.) to perform all different tasks. Therefore, the previous multi-task architectures are not optimized for such multi-task NLP processing.

Sparse GEMM Accelerators. We use sparse accelerators in our heterogeneous architecture to speed up partially shared layers. There are many sparse GEMM accelerators in the literature [12, 16, 18, 29, 31, 32, 40, 43, 44, 53–55]. Among them, OuterSPACE [29], ExTensor [18], MatRaptor [43], SpArch [55], Gamma [53] mainly aim to accelerate very sparse (0.00001-1% density) matrix multiplications in recommendation systems, computational chemistry, internet and social media applications, etc., which is not suitable for sparse neural networks (1% - 50% density). Among the sparse DNN-based accelerators, EIE [16] uses outer-product to accelerate sparse matrix vector multiplication, while SparTen [48] uses inner-product. Tensaurus [44] introduces a new sparse format, CISS, for sparse related algebra computation. SIGMA [32] uses a highly flexible dot-product engine and forward adder network to enable efficient sparse DNN. Griffin [40] describes a systematic approach to model the sparse architectures and proposes a hybrid architecture to enhance dual sparsity computation. AESPA [31] introduces a heterogeneous architecture with different sparse sub-accelerators to support various matrix sizes and sparsity. In our work, we choose SIGMA [32] for our sparse core architecture.

7 CONCLUSION

This paper presents *TaskFusion*, which to our knowledge, is the first software-hardware co-optimized architecture designed for efficient multi-task NLP. By using ℓ_0 and ℓ_1 regularization on delta weights and delta activations, respectively, our algorithm boosts data sharing between base task and sub-tasks, with negligible accuracy loss. To fully exploit delta sparsity, we propose a novel hardware-aware sub-task inference algorithm. We also propose *TaskFusion* architecture and scheduling to further accelerate the sub-task inference in a system setting. Comparing to previous single-task NLP accelerators, our method achieves 1.48-2.43× speed-up and 1.62-3.77× energy efficiency increase on average. In the long run, we believe that the development of pre-trained models can become more general and there will be more opportunities in data sharing between base task and sub-tasks.

REFERENCES

[1] Zaid Alyafeai, Maged Saeed AlShaibani, and Irfan Ahmad. 2020. A survey on transfer learning in natural language processing. *arXiv preprint arXiv:2007.04239* (2020).

[2] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. 2020. A multi-neural network acceleration architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 940–953.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[4] Daniel Cer, Mona Diab, Eneko Agirre, Inigo Lopez-Gazpio, and Lucia Specia. 2017. Semeval-2017 task 1: Semantic textual similarity-multilingual and cross-lingual focused evaluation. *arXiv preprint arXiv:1708.00055* (2017).

[5] Yujeong Choi and Minsoo Rhu. 2020. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 220–233.

[6] Ido Dagan, Oren Glickman, and Bernardo Magnini. 2005. The pascal recognising textual entailment challenge. In *Machine learning challenges workshop*. Springer, 177–190.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[8] Bill Dolan and Chris Brockett. 2005. Automatically constructing a corpus of sentential paraphrases. In *Third International Workshop on Paraphrasing (IWP2005)*.

[9] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[10] Hongxiang Fan, Thomas Chau, Stylianos I Venieris, Royson Lee, Alexandros Kouris, Wayne Luk, Nicholas D Lane, and Mohamed S Abdelfattah. 2022. Adaptable Butterfly Accelerator for Attention-based NNs via Hardware and Algorithm Co-design. *arXiv preprint arXiv:2209.09570* (2022).

[11] Cheng Fu, Hanxian Huang, Xinyun Chen, Yuandong Tian, and Jishen Zhao. 2021. Learn-to-share: A hardware-friendly transfer learning framework exploiting computation and parameter sharing. In *International Conference on Machine Learning*. PMLR, 3469–3479.

[12] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. 2019. SparTen: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 151–165.

[13] Demi Guo, Alexander M Rush, and Yoon Kim. 2020. Parameter-efficient transfer learning with diff pruning. *arXiv preprint arXiv:2012.07463* (2020).

[14] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, et al. 2020. Aˆ 3: Accelerating attention mechanisms in neural networks with approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 328–341.

[15] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W Lee. 2021. ELSA: Hardware-Software co-design for efficient, lightweight self-attention mechanism in neural networks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 692–705.

[16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

[17] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. 2022. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 16000–16009.

[18] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 319–333.

[19] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*. PMLR, 2790–2799.

[20] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[21] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. 2021. Heterogeneous dataflow accelerators for multi-DNN workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 71–83.

[22] Bingbing Li, Santosh Pandey, Haowen Fang, Yanjun Lyv, Ji Li, Jieyang Chen, Mimi Xie, Lipeng Wan, Hang Liu, and Caiwen Ding. 2020. Ftrans: energy-efficient acceleration of transformers using fpga. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. 175–180.

[23] Zheng Li, Soroush Ghodrati, Amir Yazdanbakhsh, Hadi Esmaeilzadeh, and Mingu Kang. 2022. Accelerating attention through gradient-based learned runtime pruning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 902–915.

[24] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[25] Christos Louizos, Max Welling, and Diederik P Kingma. 2017. Learning sparse neural networks through L_0 regularization. *arXiv preprint arXiv:1712.01312* (2017).

[26] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 977–991.

[27] Mary Ann Marcinkiewicz. 1994. Building a large annotated corpus of English: The Penn Treebank. *Using Large Corpora* 273 (1994).

[28] Francisco Muñoz-Martínez, José L Abellán, Manuel E Acacio, and Tushar Krishna. 2021. STONNE: Enabling cycle-level microarchitectural simulation for dnn inference accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 201–213.

[29] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.

[30] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2009), 1345–1359.

[31] Eric Qin, Raveesh Garg, Abhimanyu Bambhaniya, Michael Pellauer, Angshuman Parashar, Sivasankaran Rajamanickam, Cong Hao, and Tushar Krishna. 2022. Enabling Flexibility for Sparse Tensor Acceleration via Heterogeneity. *arXiv preprint arXiv:2201.08916* (2022).

[32] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 58–70.

[33] Zheng Qu, Liu Liu, Fengbin Tu, Zhaodong Chen, Yufei Ding, and Yuan Xie. 2022. DOTA: detect and omit weak attentions for scalable transformer acceleration. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 14–26.

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

[35] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[36] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[37] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2018. SCALE-Sim: Systolic CNN Accelerator Simulator. *arXiv preprint arXiv:1811.02883* (2018).

[38] Erik F Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. *arXiv preprint cs/0306050* (2003).

[39] Erik F Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. *arXiv preprint cs/0306050* (2003).

[40] Jong Hoon Shin, Ali Shafiee, Ardavan Pedram, Hamzah Abdel-Aziz, Ling Li, and Joseph Hassoun. 2022. Griffin: Rethinking Sparse Optimization for Deep Learning Architectures. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 861–875.

[41] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*. 1631–1642.

[42] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*. 1631–1642.

[43] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[44] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 689–702.

[45] Thierry Tambe, Coleman Hooper, Lillian Pentecost, Tianyu Jia, En-Yu Yang, Marco Donato, Victor Sanh, Paul Whatmough, Alexander M Rush, David Brooks, et al. 2021. Edgebert: Sentence-level energy optimizations for latency-aware multi-task nlp inference. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 830–844.

[46] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (1996), 267–288.

[47] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* (2018).

[48] Hanrui Wang, Zhekai Zhang, and Song Han. 2021. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 97–110.

[49] Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. 2019. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics* 7 (2019), 625–641.

[50] Adina Williams, Nikita Nangia, and Samuel R Bowman. 2017. A broad-coverage challenge corpus for sentence understanding through inference. *arXiv preprint arXiv:1704.05426* (2017).

[51] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. 2020. Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 811–824.

[52] Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. 2021. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. *arXiv preprint arXiv:2106.10199* (2021).

[53] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 687–701.

[54] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[55] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.

[56] Zhe Zhou, Junlin Liu, Zhenyu Gu, and Guangyu Sun. 2022. Energon: Towards Efficient Acceleration of Transformers Using Dynamic Sparse Attention. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022).