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  
niruizh@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 $\ell_1$ 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.

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:


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.
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 BERT/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 $I_{th}$ linear layer.

Layer $L$ of the base task:

$$A^{L+1}_{base} = A^L_{base} \times W^L_{base}$$  \hspace{1cm} (1)

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

$$A^{L+1}_{sub_F} = A^L_{sub_F} \times W^L_{sub_F}$$  \hspace{1cm} (2)

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

$$A^{L+1}_{sub_P} = A^L_{sub_P} \times (W^L_{base} + \delta W^L_{sub_P})$$  \hspace{1cm} (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 $\delta W_{sub_P}$ 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
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:

- 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 2.86x 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 \( \mathcal{D}_{\tau} = \{x^{(n)}_\tau, y^{(n)}_\tau\}_{n=1}^N \). For all tasks, task-specific fine-tuning aims to produce a set of model parameters \( w_\tau \) to optimize the following constrained optimization problem:

\[
\min_{w_\tau} \frac{1}{N} \sum_{i=1}^{N} C(f(z^{(n)}_\tau; w_\tau), y^{(n)}_\tau) + \lambda R(\cdot) \quad (4)
\]

where \( f_\tau \) is a neural network function, \( C(\cdot) \) is a cost function (e.g., cross-entropy) and \( R(\cdot) \) is an optional regularization function with hyperparameter \( \lambda \).

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 2: Delta activation density rises rapidly even in the first linear layer.](image)

![Figure 3: Pre-trained model and different weight tuning methods.](image)
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 sub-layers 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 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.

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:

$$A_{L+1}^{\text{sub-T}} = A_{L}^{\text{base}} \times W_{L}^{\text{base}} = A_{L}^{\text{base}}$$  \hspace{1cm} (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,
and then the activation during sub-task processing can be written as
\[ A_{\text{sub}_T}^{L+1} = A_{\text{sub}_T}^{L} \times W_{\text{sub}_T}^{L} \]
where \( T \) stands for TaskFusion. \( \delta W_{\text{sub}_T}^{L} \) and \( \delta A_{\text{sub}_T}^{L} \) 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**
  \[ N_{TS} \leq N < N_{TS} + N_{PS}; \]
  \[ A_{\text{sub}_T}^{L+1} = A_{\text{sub}_T}^{L} \times W_{\text{sub}_T}^{L} \]
  \[ = (A_{\text{base}}^{L} + \delta A_{\text{sub}_T}^{L}) \times (W_{\text{base}}^{L} + \delta W_{\text{sub}_T}^{L}) \]
  \[ = A_{\text{base}}^{L} \times W_{\text{base}}^{L} + \delta A_{\text{sub}_T}^{L} \times (W_{\text{base}}^{L} + \delta W_{\text{sub}_T}^{L}) \]
  \[ + A_{\text{base}}^{L} \times \delta W_{\text{sub}_T}^{L} \]
  \[ = A_{\text{base}}^{L+1} + \delta A_{\text{sub}_T}^{L} \times W_{\text{sub}_T}^{L} + A_{\text{base}}^{L} \times \delta W_{\text{sub}_T}^{L} \]

Since \( \delta W_{\text{sub}_T}^{L} \) and \( \delta A_{\text{sub}_T}^{L} \) are sparse matrices, it is no longer necessary to perform a new dense matrix-matrix multiplication \( (A_{\text{sub}_T}^{L}) \times W_{\text{sub}_T}^{L} \). Instead, the pre-computed base task activation \( A_{\text{base}}^{L} \) can be reused in sub-task computation as shown in Eq. 6. By this sharing, two sparse-dense matrix-matrix multiplications \( (\delta A_{\text{sub}_T}^{L} \times W_{\text{sub}_T}^{L}) \) replace the dense matrix-matrix multiplication. When \( \delta A_{\text{sub}_T}^{L} \) and \( \delta W_{\text{sub}_T}^{L} \) 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:

\[ \delta A_{\text{sub}_T}^{L+1} = \text{topK}(\delta A_{\text{sub}_T}^{L} \times W_{\text{sub}_T}^{L} + A_{\text{base}}^{L} \times \delta W_{\text{sub}_T}^{L}) \]

In non-shareable layers, activations are not shared but the delta weight matrix \( \delta W_{\text{sub}_T}^{L} \) 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**
  \[ L \geq N_{TS} + N_{PS}; \]
  \[ A_{\text{sub}_T}^{L+1} = A_{\text{sub}_T}^{L} \times W_{\text{sub}_T}^{L} \]
  \[ = A_{\text{base}}^{L} \times W_{\text{base}}^{L} + \delta A_{\text{sub}_T}^{L} \times \delta W_{\text{sub}_T}^{L} \]  

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 \( \delta W_{\text{sub}_T}^{L} \) and \( \delta A_{\text{sub}_T}^{L} \) 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 \( w_s \) and activation is \( a_s \). We define the delta weight and delta activation as:

\[ \delta w_s = w_s - w_p, \delta a_s = a_s - a_p \]

In order to make \( \delta w_s \) and \( \delta a_s \) sparse, we modify the optimization loss function from Eq.4 to:

\[ \min w_s \frac{1}{N} \sum_{i=1}^{N} C(f_T(x_{T}^{(i)}; w_s), y_{T}^{(i)}) + R_w(\delta w_s) + R_d(\delta a_s) \]  

Since diff pruning has proven the effectiveness of \( \ell_0 \) regularization on sparsifying \( \delta w_s \), we use the same method to learn the sparse \( \delta w_s \) by using:

\[ R_w(\delta w_s) = \lambda_w \sum_{i=0}^{L} ||\delta w_s^{L}||_0 = \lambda_w \sum_{i=0}^{L} \sum_{j=1}^{d} \mathbb{1}\{ \delta w_s^{L,j} \neq 0 \} \]

where \( L \) equals the total number of layers and \( \lambda_w \) is the regularization 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 \( \delta w_s \) sparse.

Interestingly, the same approach does not extend well to sparse delta activation learning. Since delta activation \( \delta a_s \) also depends on the variable input \( x_r \), it is hard to learn a fixed deterministic binary mask applicable to all inputs. \( \ell_1 \) 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 \( \ell_1 \) regularization on delta activations during training to shape the distribution of \( \delta a_s \) to have a higher probability of small values. The proposed regularization function of \( \delta a_s \) is written as:

\[ R_d(\delta a_s) = \sum_{i=0}^{L} \lambda_a^{d} ||\delta a_s^{L}||_1 = \sum_{i=0}^{L} \lambda_a^{d} \sum_{j=1}^{d} |\delta a_s^{L,j}| \]

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

Fig. 7 shows the \( \delta a_s \) distribution of BERTbase model 4th and 5th layer on sentiment analysis (SST-2) task with (a) no regularization, (b) \( \ell_1 \) regularization, and (c) top-K selection after (b) (top 20% selected). Note that by using \( \ell_1 \) 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 \( \delta a_s \) and enables sparse matrix multiplications to replace dense multiplications.
fun related activation is left and passed to the next layer. By adding ISCA and is, are identified as unimportant tokens. Therefore, only with the tokens “ISCA” and “is” still pruned out. This combination, base task and sub-tasks partially share “fun” activations, makes the delta activation smaller and also sparser. Experiments show only negligible accuracy loss with this combination (Sec. 5).

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
- 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. 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. 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
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) 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).

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 \( \delta W_{\text{sub}_T} \) and \( \delta a_{\text{sub}_T} \) are chosen dynamically based on their magnitudes. There have been many sparse DNN accelerators [12, 16, 18, 31, 40, 43, 44, 53, 54] in recent years to accelerate sparse matrix multiplications. We choose SIGMA-like [32] architecture (Fig. 9) 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 \( \delta A_{\text{sub}_T} \times W_{\text{sub}_T} \). 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 \( A_{\text{base}} \times \delta W_{\text{sub}_T} \) that reads \( \delta W_{\text{sub}_T} \) from the sub-task delta weight memory and \( A_{\text{base}} \) 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 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 \( W_{\text{base}}^L + \delta W_{\text{sub}_T}^L \) in Eq. 6, we design a sparse-dense addition unit in the weight memory system. First, the

![Figure 9: TaskFusion architecture.](image-url)
non-zero delta weight positions are identified and stored while computing $A^t_{\text{base}} \times \delta W^{\text{sub}}_t$. 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 ($W^{\text{Vmem}}_{\text{base}}$) are read from block1, and the sparse core generates sub-task $K$. The base K weights ($W^{\text{Kmem}}_K$) are read from block2, which are added to sub-task (11 or 12) delta K weights ($\delta W^{\text{Kmem(sub)}}_{\text{sub}_1/2}$) and sent to the sparse core. Meanwhile, block3 receives the next layer’s weight ($W^{\text{Vmem}}_{\text{base}}$) from DRAM. For the next stage (Fig. 10 right), the sparse core can reuse $W^{\text{Vmem}}_{\text{base}}$ from block1. Since $W^{\text{Kmem}}_{\text{base}}$ is no longer useful, block2 is updated with next layer’s weights ($W^{\text{Fmem}}_{\text{sub}_1/2}$). This rotation eliminates on-chip weight access conflicts and also minimizes off-chip weight accesses from DRAM.

The activation memory system (Fig. 9 3) 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: BERT$^{\text{base}}$ and BERT$^{\text{large}}$. 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 BERT\textsubscript{base} / BERT\textsubscript{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 Verilogs. We use SIGMA’s open-source RTL [32] for sparse core energy simulations. All Verilogs are synthesized using Synopsys Design Compiler in TSMC 22nm ULL technology at the frequency of 1GHz and we use PrimePower with FSBM wave- form 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, $\lambda_0$ regularization on delta weights and $\ell_1$ 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, $\ell_1$ regularization coefficient ($\lambda_1$, 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 (BERT\textsubscript{base} and BERT\textsubscript{large}) on 10 sub-tasks. For BERT\textsubscript{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 BERT\textsubscript{large} **TaskFusion** can save 65.2% / 62.4% FLOPs on average for BERT\textsubscript{base} / BERT\textsubscript{large} while SpAtten + **TaskFusion** increases FLOPs saving to 73.1% / 69.1% for BERT\textsubscript{base} / BERT\textsubscript{large} compared to the baseline.

We conducted additional experiments to show the benefits of $\ell_1$ regularization on delta activations. Fig. 13 shows the accuracy of sub-task SST-2 and QNLI in **TaskFusion** (no SpAtten) with varying layer-wise $\ell_1$ regularization coefficients $\lambda_1$ (refer to Eq.10) when the target delta activation sparsity is set to 20%. $\lambda_0 = 0$ means there is no $\ell_1$ penalty on delta activations. BERT\textsubscript{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 $\ell_1$ regularization causes large accuracy drop (> 2%). Using a small $\ell_1$ regularization coefficient improves accuracy. However, when $\lambda_0$ is too large, accuracy decreases. In our training for different sub-tasks, we set $\lambda_0$ 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>
<thead>
<tr>
<th>MRPC</th>
<th>STS-B</th>
<th>RTE</th>
<th>SST-2</th>
<th>QNLI</th>
<th>MNLI</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline Acc.</td>
<td>92.2%</td>
<td>98.8%</td>
<td>90.1%</td>
<td>91.2%</td>
<td>91.9%</td>
<td>87.8%</td>
</tr>
<tr>
<td>TaskFusion Acc.</td>
<td>91.3%</td>
<td>99.3%</td>
<td>78.5%</td>
<td>94.0%</td>
<td>91.7%</td>
<td>86.8%</td>
</tr>
<tr>
<td>FLOPs reduction</td>
<td>54%</td>
<td>99%</td>
<td>64%</td>
<td>72%</td>
<td>59%</td>
<td>76%</td>
</tr>
</tbody>
</table>

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:
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-task. 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 BERTbase and BERTlarge. The average sub-task speed up with TaskFusion is 2.85× for BERTbase and 2.69× for BERTlarge. 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 BERTbase and 2.96× for BERTlarge compared to only using SpAtten, and 4.16× for BERTbase and 3.35× for BERTlarge compared to the baseline architecture.

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 0.06mm² (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 BERTbase and BERTlarge. In summary, TaskFusion achieves 141.4× / 14.2×
Table 2 shows the comparison between 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 BERTbase 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 BERT\textsubscript{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

---

**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.
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 2.52x system speed-up). By adding scheduling, the total system latency reduction increases up to 67.7% (equals to 3.10x 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].

### 6 RELATED WORKS

#### Hardware Support for Transformers
Many hardware accelerators have recently been proposed for accelerating transformer-like neural networks. $\text{A}^3$ [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, Sanger [26] 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

![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.](image-url)
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, 29, 31, 32, 40, 43, 44, 53–55]. Among them, OuterSPACE [29], ExTensor [18], MatRaptor [45], SwArch [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.

**REFERENCES**


