MCUBERT: Memory-Efficient BERT Inference on Commodity Microcontrollers

Zebin Yang[1,2†], Renze Chen[3†], Taiqiang Wu[4], Ngai Wong[4], Yun Liang[2,6], Runsheng Wang[2,5,6], Ru Huang[2,5,6], Meng Li[1,2,6*]

[1]Institute for Artificial Intelligence, Peking University, Beijing, China
[2]School of Integrated Circuits, Peking University, Beijing, China
[3]School of Computer Science, Peking University, Beijing, China
[4] The University of Hong Kong, Hong Kong, China
[5]Institute of Electronic Design Automation, Peking University, Wuxi, China
[6]Beijing Advanced Innovation Center for Integrated Circuits, Beijing, China

ABSTRACT

In this paper, we propose MCUBERT to enable language models like BERT on tiny microcontroller units (MCUs) through network and scheduling co-optimization. We observe the embedding table contributes to the major storage bottleneck for tiny BERT models. Hence, at the network level, we propose an MCU-aware two-stage neural architecture search algorithm based on clustered low-rank approximation for embedding compression. To reduce the inference memory requirements, we further propose a novel fine-grained MCU-friendly scheduling strategy. Through careful computation tiling and re-ordering as well as kernel design, we drastically increase the input sequence lengths supported on MCUs without any latency or accuracy penalty. MCUBERT reduces the parameter size of BERT-tiny and BERT-mini by 5.7× and 3.0× and the execution memory by 3.5× and 4.3×, respectively. MCUBERT also achieves 1.5× latency reduction. For the first time, MCUBERT enables lightweight BERT models on commodity MCUs and processing more than 512 tokens with less than 256KB of memory.

KEYWORDS

MCU, BERT, MCU-aware NAS, MCU-friendly Scheduling Optimization, Custom Kernel Design

1 INTRODUCTION

IoT devices based on microcontroller units (MCUs) are ubiquitous, enabling a wide range of speech and language applications on the edge, including voice assistant [9, 38], real-time translation [40, 45], smart home [35], etc. Language models (LMs), e.g., BERT [13], are fundamental to these applications. While cloud off-loading is heavily employed for LM processing, it suffers from high latency overhead, privacy concerns, and a heavy reliance on WiFi or cellular networks [3, 16]. Hence, there is a growing demand for deploying BERT-like LMs on MCUs.

*Corresponding Author, meng.li@pku.edu.cn
†Equal contribution

ICCAD '24, October 27–31, 2024, New York, NY, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1077-3/24/10…$15.00
https://doi.org/10.1145/3676536.3676747

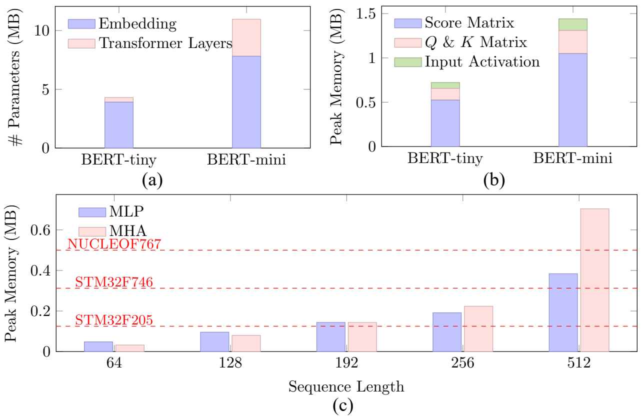


Figure 1: Enabling BERT on MCUs faces memory challenges: (a) the Flash storage limits the model size; (b) the SRAM memory limits the peak execution memory; (c) for long sequence lengths, memory requirements of both MHA and multi-layer perceptron (MLP) become bottleneck.

Though appealing, enabling BERT on MCUs is very challenging. On one hand, MCUs only have a very simple memory hierarchy with highly constrained memory budgets [3, 5]. For example, a state-of-the-art (SOTA) ARM Cortex-M7 MCU only has 320 KB static random-access memory (SRAM) to store intermediate data, e.g., activation, and 1 MB Flash to store program and weights, directly limiting the peak execution memory and the total parameter size of LMs [3, 5, 23, 24]. As shown in Figure 1, even with 8-bit quantization, the BERT-tiny model [4, 36] exceeds the SRAM and Flash capacity by more than 2.3× and 4.3×, respectively.

On the other hand, the computation graph of a Transformer block in BERT is more complex when compared with convolutional neural networks (CNNs): each multi-head attention (MHA) block not only comprises more MCU-unfriendly tensor layout transformation operators, e.g., reshape, transpose, etc, but also organizes them with linear operators in a more complex topology. Naively executing these operators can result in significant memory consumption and poor performance [5, 18].

Existing works, however, cannot resolve these challenges. Most SOTA BERT designs [19, 21, 32] and network optimization designs

Table 1: Comparison with prior-art methods (Opt represents Optimization).

Method	Platform	Model	Network Opt	Scheduling Opt		
				MLP	MHA	Kernel
[3]	MCU	CNN	Conv	-	-	✗
[23, 24]	MCU	CNN	Conv	-	-	✓
[22]	MCU	ViT	MLP	✓	✗	✓
[32]	GPU	BERT	Linear	✗	✗	✗
[11]	GPU	BERT	-	✗	✓	✓
Ours	MCU	BERT	Embedding	✓	✓	✓

[39, 44] focus on large BERT models on GPUs or smartphones. As the memory and storage of these platforms are much more abundant than MCUs, these methods usually focus on optimizing the Transformer layers for better latency. Another class of works [3, 15, 23, 24] develop MCU-friendly CNNs by jointly optimizing the network architecture and scheduling, which, however, cannot be directly applied to BERT.

In this paper, we propose MCUBERT, a network and scheduling co-optimization framework to enable BERT on commodity MCUs for the first time. We observe for small BERT models, e.g., BERT-tiny, the embedding table accounts for the major storage bottleneck and thus, propose to leverage the MCU-friendly clustered low-rank approximation for embedding compression. We further propose an MCU-aware two-stage differentiable neural architecture search (NAS) algorithm to improve the accuracy of compressed models. To reduce the execution memory usage and latency, we observe the execution of a Transformer block can be carefully tiled without accuracy penalty and develop an MCU-friendly fine-grained scheduling algorithm. Our contributions can be summarized as follows:

- We propose an MCU-aware two-stage NAS algorithm based on clustered low-rank approximation for embedding compression to overcome the storage bottleneck.
- To reduce the execution peak memory and latency, we propose MCU-friendly scheduling optimization for the Transformer block.
- MCUBERT can reduce the model size by 5.7× and 3.0× as well as the peak memory by 3.5× and 4.3× for BERT-tiny and BERT-mini, respectively, enabling to process more than 512 tokens simultaneously with less than 256KB SRAM. MCUBERT also achieves 1.5× and 1.3× latency reduction compared to SOTA inference engines, CMSIS-NN and [5], respectively.

2 RELATED WORKS

2.1 Model Deployment on MCUs.

There are two main approaches to deploy models on MCUs, interpretation [8, 12] and code generation [5, 23, 24]. The interpretation-based methods embed an on-device runtime interpreter to support flexible model deployment but require extra memory to store meta-information and extra time for runtime interpretation. Code-generation-based methods directly compile the given model into target code to save memory and reduce inference latency. MCUBERT uses a code generation method that is more specialized for our target model and device, leading to lower latency and lower memory usage.

2.2 Network efficiency optimization.

Network efficiency is very important for the overall performance of deep learning systems and has been widely studied. We focus on reviewing the model optimizations targeting at MCUs and for LMs.

Deep learning models need to meet the tight storage and memory constraints to run on MCUs. Previous works propose network and scheduling optimization as summarized in Table 1. [3, 24, 31] compress CNNs with NAS to meet the storage constraints. [5] deploys small Transformer models which already satisfy the memory constraints of MCUs. [22] deploys vision transformer (ViT) on MCUs mainly by compressing the MLP layers and searching the token numbers. However, there are new challenges for deploying BERT on MCUs. First, compared to tiny CNNs and small Transformer models, BERT models have more parameters. Second, the memory bottleneck of BERT inference mainly lies in the MHA block when sequence length is long, which is shown in Figure 1(c). The computation of MHA block is more complex compared with CNN blocks and MLP layers, bringing new challenges for scheduling optimization. We propose MCUBERT to first deploy BERT, the most representative encoder transformer model, on MCUs.

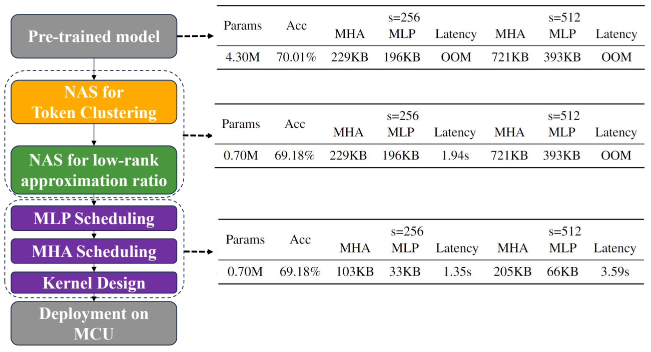
Though no existing works deploy BERT models on MCUs, there are network optimization algorithms proposed for LMs on other platforms, e.g., GPUs, such as pruning [6, 10, 34, 41], quantization [2, 28, 33, 43], low-rank factorization [1, 7, 17, 21, 25], etc. As MCUs do not natively support low-bit quantization or sparse computation, low-rank factorization is usually more MCU-friendly. Existing works such as Distilled Embedding [25] and Albert [21] leverage low-rank factorization based on singular value decomposition (SVD) for embedding compression and prune singular values with small magnitudes. Adaptive embedding [1] compresses the embedding table with clustered low-rank approximation: it divides the tokens in an embedding table into clusters first and applies low-rank approximation with different ratios to each cluster based on the cluster importance. Adaptive embedding empirically leverages token frequency as the proxy metric for its importance, which leads to high accuracy degradation.

There are also scheduling optimization proposed for Transformer models on GPUs. FlashAttention [11] only computes the attention score tensor partially each time and repetitively update the accumulation of the partial sum to drastically reduce the execution memory. There are also previous works like [14, 30, 42] that use kernel fusion to reduce memory usage and accelerate inference. However, these methods usually do not consider quantization and targets at both training and inference. Our scheduling optimization is inspired by these methods but is more MCU-friendly.

As shown in Table 1, our proposed MCUBERT first deploys transformer model BERT on MCUs. MCUBERT compresses the embedding table, which account for most of the parameters and can't be stored in MCU. Besides MLP block, We also carefully optimize the MHA block, which comprises more MCU-unfriendly operators and can also be the memory bottleneck of BERT inference. Compared with GPU-based methods, MCUBERT conduct network and scheduling optimization in a more MCU-friendly mode.

Table 2: Notations used in the paper.

Notations	Meanings
v	Vocabulary size
s, h, d	Sequence length, # heads, and embedding dimension
c	# clusters
i, j, l	Loop variables for clusters, tokens, and singular values
t	Sequence length each tile
U, V	Unitary matrices generated by SVD
U_i, V_i	Embedding table and linear projection for the i_{th} cluster
$U_{i,j}$	Embedding vector for the j_{th} token in i_{th} cluster
Σ	Vector of singular values
α, β	NAS parameters for embedding compression
$\alpha_{j,i}$	NAS parameters for j_{th} token in i_{th} cluster in first stage NAS
$\beta_{i,l}$	NAS parameters for l_{th} singular value in i_{th} cluster in second stage NAS
β_i^*	Threshold for NAS parameters in i_{th} cluster in second stage NAS
M, N, K	Loop ranges of a matrix multiplication
m, n, k	Loop variables of a matrix multiplication



Params	Acc		s=256			s=512	
		MHA	MLP	Latency	MHA	MLP	Latency
4.30M	70.01%	229KB	196KB	OOM	721KB	393KB	OOM

Params	Acc		s=256			s=512	
		MHA	MLP	Latency	MHA	MLP	Latency
0.70M	69.18%	229KB	196KB	1.94s	721KB	393KB	OOM

Params	Acc		s=256			s=512	
		MHA	MLP	Latency	MHA	MLP	Latency
0.70M	69.18%	103KB	33KB	1.35s	205KB	66KB	3.59s

Figure 2: MCUBERT overview. (Params stands for parameters, Acc stands for MNLI accuracy, and OOM stands for out of memory.)

3 MCUBERT: MCU-FRIENDLY NETWORK/SCHEDULING CO-OPTIMIZATION

3.1 Motivations and Overview

We now discuss our observations on key challenges that prevent running BERTs directly on MCUs and introduce the overall flow of MCUBERT. The notations used in the section is summarized in Table 2.

Observation 1: the tight MCU Flash storage limits model size and forces to use small BERT models, for which the embedding table becomes the major bottleneck. To satisfy the Flash storage constraints of MCUs, which is often less than 2 MB [3, 24], we are forced to consider only small BERT models, e.g., BERT-tiny and BERT-mini [13], which have lower embedding dimensions and fewer Transformer layers. However, there still exists a 4.3× gap between the model size and the MCU Flash, even for BERT-tiny with 8-bit quantization. As shown in Figure 1(a), the embedding table contributes to more than 90% of the parameters of BERT-tiny and becomes the major bottleneck. Hence, embedding compression is required to enable BERT on MCUs.

Observation 2: the MCU SRAM size limits the execution peak memory of both MHA and MLP, especially for long sequence lengths. During BERT inference, all the activations need to be stored in the MCU SRAM. Although Flash paging and rematerialization has been proposed [27] to reduce the SRAM requirements, the introduced Flash access and re-computation incur high power and latency overhead. As shown in Figure 1(c), with the increase of the sequence length, the peak memory of both MHA and MLP increases significantly and quickly exceed the MCU SRAM limit. Depending on the sequence length, both MLP and MHA can be the memory bottleneck. Meanwhile, as shown in Figure 1(b), for a long input sequence, the score matrix in MHA incurs highest memory consumption. Therefore, both MLP and MHA, especially the score matrix, needs to be optimized to enable processing long sequences.

Observation 3: the naive design of computation kernels introduces non-negligible latency overhead. Both MHA and MLP involve many linear/batched matrix multiplication operators with large shapes. MHA further complicates the inference with extensive tensor shape transformation operators, e.g., reshape, transpose, etc. Naive kernel implementation, e.g., CMSIS-NN, cannot well utilize the hardware resource and suffers from over 30% latency overhead. To improve the resource utilization, it is important to leverage the memory locality and the instruction-level parallelism (ILP) of MCU. Memory access patterns can also be designed to fuse the tensor shape transformation operators with the linear operators. All of these require dedicated kernel optimization.

MCUBERT overview. Based on these observations, we propose MCUBERT, an MCU-friendly network/scheduling co-optimization framework to enable BERT models on tiny devices. The overview of MCUBERT is shown in Figure 2. MCUBERT leverages MCU-friendly clustered low-rank approximation for embedding compression and features a MCU-aware two-stage NAS to explore the trade-off between accuracy and model sizes, enabling to reduce the model size and fit BERT models into the MCU Flash storage. To enable processing long sequences, MCUBERT proposes MCU-friendly scheduling optimization. By tiling and re-ordering the computation as well as designing efficient kernels, inference peak memory and latency can be reduced without accuracy penalty. Such optimization enables to support more than 512 tokens on a small MCU with less than 256 KB SRAM.

Note while we focus on optimizing BERT-like encoder LMs for MCUs, our proposed techniques, including embedding compression and scheduling optimization, can benefit GPT-like decoder LMs [29] on MCUs as well. Decoder LMs face more challenges on the storage and dynamic shape induced by the KV cache, which we leave for future research.

3.2 MCU-aware NAS for Embedding Compression

MCU-aware NAS Formulation. To reduce the parameter size and satisfy the tight Flash storage constraint, we propose embedding compression based on clustered low-rank approximation following [1], which is more MCU-friendly as MCUs do not natively support sub-8-bit quantization or sparse computation. Following

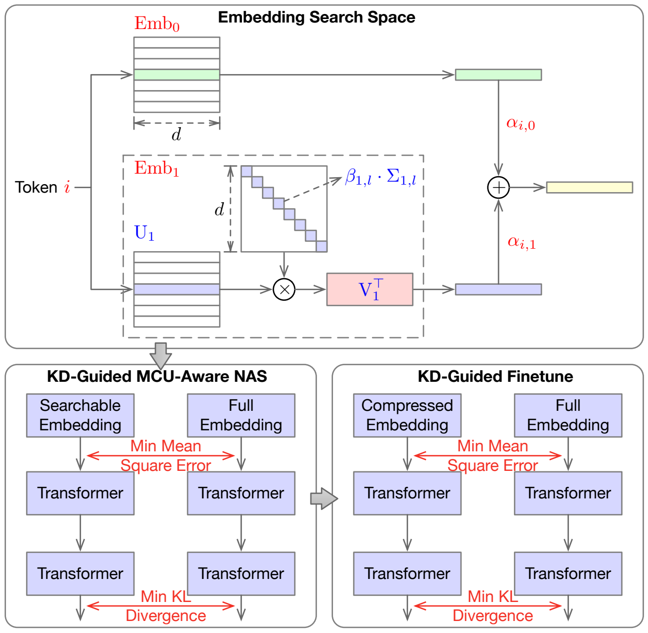


Figure 3: Our proposed NAS search formulation for embedding compression.

[1], we set the number of clusters to 4 and show its impact in ablation study. Two important questions remain to be answered: 1) how to cluster the tokens and 2) how to determine the approximation ratio for each cluster. We propose differentiable NAS to guide the compression directly with task loss.

Let c denote the number of clusters and Emb_i denote the embedding table for i-th cluster, where $0 \leq i \leq c-1$. To enable NAS for token clustering, as shown in Figure 3, we introduce trainable architecture parameters α $(0 \leq \alpha \leq 1)$ for each token in the vocabulary. Let v denote the vocabulary size. Then, for the j-th token $(0 \leq j \leq v-1)$, we define $\{\alpha_{j,0}, \ldots, \alpha_{j,c-1}\}$, where $\sum_{i=0}^{c-1} \alpha_{j,i} = 1$. Then, the embedding for the j-th token becomes:

$$\alpha_{j,0}\text{Emb}_0[j] + \sum_{i=1}^{c-1} \alpha_{j,i}\text{Emb}_i[j], \tag{1}$$

where $\text{Emb}_i[j]$ denotes the embedding vector of j-th token from the i-th embedding table.

To leverage NAS for low-rank approximation, instead of directly searching the reduced dimension for each embedding table, we apply singular value decomposition (SVD) and search the singular values that can be pruned, which enables to better inherit the pre-trained weights and is crucial for BERT compression. Specifically, we first decompose Emb_i with SVD as $U_i\text{Diag}(\Sigma_i)V_i^\top$, where U_i and V_i are unitary matrices, Σ_i is the vector of singular values, and $\text{Diag}(\Sigma_i)$ represents the diagonal matrix with singular values in Σ_i filled in the diagonal. We also introduce architecture parameters β $(0 \leq \beta \leq 1)$ for each cluster. For the i-th cluster, we define $\{\beta_{i,0}, \ldots, \beta_{i,d-1}\}$, where d is the embedding dimension, to indicate the importance of each singular value. Then, we have

$$\text{Emb}_i = U_i\text{Diag}(\beta_{i,0}\Sigma_{i,0}, \ldots, \beta_{i,d-1}\Sigma_{i,d-1})V_i^\top. \tag{2}$$

Based on α and β, the embedding size for the i-th cluster can be approximated as

$$\text{Size}(\text{Emb}_i) = \sum_l \beta_{i,l}(d + \sum_j \alpha_{j,i}), \tag{3}$$

where $\sum_j \alpha_{j,i}$ approximates the number of tokens for i-th cluster. $\sum_l \beta_{i,l} \sum_j \alpha_{j,i}$ and $\sum_l \beta_{i,l} d$ represent the size of low-rank approximated U_i and $V_i^\top$, respectively. We introduce ℓ_1-penalty of the embedding table size into the objective function to encourage a lower parameter size:

$$\min_{\alpha,\beta} \min_{w} \ell_{w,\alpha,\beta} + \lambda \sum_i \text{Size}(\text{Emb}_i). \tag{4}$$

where w denotes the model parameters and λ is the hyperparameter to balance model size and accuracy. α, β, and w are trained together. Upon convergence, for the j-th token, we determine its cluster by $\text{argmax}_i \alpha_{j,i}$, while for the i-th cluster, we set a threshold β_i^* and only keep the l-th singular values with $\beta_{i,l} > \beta_i^*$.

Improving NAS Convergence. In practice, we find that searching token clustering and low-rank approximation ratio, i.e., α and β, simultaneously makes the differentiable NAS training unstable and directly degrades the accuracy of the searched models. As shown in Figure 7, the accuracy of the searched models is even lower compared to the baseline adaptive embedding [1]. We hypothesize this is because of the large discrepancy of optimal low-rank approximation ratios for different clusters. Because the token clustering keeps changing during NAS, some singular values in a certain cluster may be incorrectly preserved due to tokens that eventually belong to a different cluster. These incorrectly preserved singular values in turn may impact the token clustering, leading to sub-optimal results.

To encourage the NAS convergence, we decompose the search space and propose a two-stage NAS strategy: in the first stage, we fix different low-rank approximation ratios for each cluster, and only search the token clustering, i.e., α. The approximation ratios are chosen following [1]. Then, in the second stage, given the fixed token clustering, we search the best approximation ratio for each cluster, i.e., β. While we can iterate back to the first stage with updated approximation ratios, we empirically find it is not necessary. We hypothesize this is because the token clustering reflects the token importance, which should remain stable for different low-rank approximation ratios. The algorithm details are shown in Algorithm 1, 2, and 3. We guide the NAS with knowledge distillation (KD) to improve convergence following [32]. We simply use the trained full model as the teacher.

Our proposed two-stage NAS incurs low training cost as only a few training epochs (less than 2) are needed for each stage. After the two-stage NAS, we quantize both the weights and activations to 8 bits and fine-tune the compressed model with KD to get the final deployable model.

3.3 MCU-friendly Scheduling Optimization

We now introduce our MCU-friendly scheduling optimization to reduce the inference memory and latency. We optimize the scheduling of both MHA and MLP since and also optimize the kernel implementation for MHA for better efficiency.

Algorithm 1 Searching for Token Clustering

```
Input: Embedding matrix $Emb$, vocabulary size $v$, # clusters $c$, embedding dimension $d$, division value $div$, # training epochs $epochs$
Output: Clustering results $cls$
1:  $\alpha_{j,i} = \frac{1}{c} \quad \forall j, i$
2:  $U_0 = Emb$
        ▷ The embedding table of first cluster is not factorized
3:  for $i = 1, \ldots, c-1$ do
4:      $U_i, V_i^T$ = LowRankFactorization($Emb$, $\frac{d}{div^i}$)
        ▷ Factorize the embedding table of other clusters except the first cluster
5:  end for
6:  Define the embedding of $j$-th token: $\alpha_{j,0} U_{0,j} + \sum_{i=1}^{c-1} \alpha_{j,i} U_{i,j} V_i^T$
        ▷ $V_0$ is excluded as the embedding table of the first cluster is not factorized
7:  for $z = 0, \ldots, epochs-1$ do
8:      Fix $\alpha$ and update weights $w$ by descending $\nabla_w \ell(w, \alpha)$
9:      Fix $w$ and update architecture parameters $\alpha$ by descending $\nabla_\alpha(\ell(w, \alpha) + \lambda \sum_i \mathrm{Size}(Emb_i))$
10: end for
11: $cls[j] = \arg\max_i(\alpha_{j,i}) \quad \forall j$
12: return $cls$
```

Algorithm 2 LowRankFactorization

```
Input: Matrix to be factorized $Matrix$; Factorization ratio $r$
Output: Factorization Results $U$ and $V^T$
1:  $U, \mathrm{Diag}(\Sigma), V^T = \mathrm{SVD}(Matrix)$
        ▷ Diag($\Sigma$) denotes the diagonal matrix filled with vector $\Sigma$
2:  $\Sigma = \Sigma[0 : r]$
3:  $U = U[:, 0 : r]\ \mathrm{Diag}(\Sigma^{1/2})$
4:  $V^T = \mathrm{Diag}(\Sigma^{1/2})\ V^T[0 : r, :]$
5:  return $U, V^T$
```

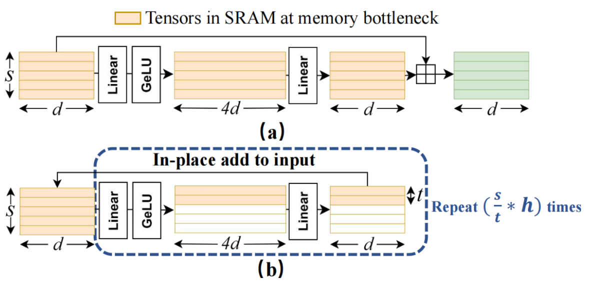


Figure 4: MLP scheduling to reduce peak memory. The tensor in yellow will be saved in SRAM at memory bottleneck.

MLP Scheduling. The memory bottleneck of MLP comes from the second linear layer. As shown in Figure 4(a), 3 tensors need to be stored, whose sizes add up to $6 \cdot s \cdot d$. We observe the computation of different tokens in an MLP is independent. This enables us to divide the input activation into tiles along the token dimension. Assume each tile has t tokens. Then, we can re-order the computation to finish all the MLP computations for one tile before moving to the next tile. Moreover, we can also perform an in-place addition with

Algorithm 3 Searching for Low-Rank Approximation Ratio

```
Input: Embedding matrix $Emb$, vocabulary size $v$, # clusters $c$, embedding dimension $d$, # training epochs $epochs$, importance threshold $\beta^*$, searched token clustering $cls$
Output: Low-rank approximation ratios $ratios$
1:  $\beta_{i,l} = \beta_i^* \quad \forall i, l$
2:  Divide the $Emb$ into $c$ parts $Emb_0, \ldots, Emb_{c-1}$ according to token clustering $cls$
3:  $U_0 = Emb_0$
        ▷ The embedding table of first cluster is not factorized
4:  for $i = 1, \ldots, c-1$ do
5:      $U_i, \mathrm{Diag}(\Sigma_{i,0}, \ldots, \Sigma_{i,d-1}), V_i^T = \mathrm{SVD}(Emb_i)$
        ▷ Factorize the embedding table of other clusters except the first cluster
6:  end for
7:  Define embedding of $j$-th token in $i$-th cluster (for $i \geq 1$): $U_{i,j} \cdot \mathrm{Diag}(\beta_{i,0}\Sigma_{i,0}, \ldots, \beta_{i,d-1}\Sigma_{i,d-1}) V_i^T$
8:  For the first cluster, the embedding of $j$-th token is $U_{0,j}$
9:  for $z = 0, \ldots, epochs-1$ do
10:     Fix $\beta$ and update weights $w$ by descending $\nabla_w \ell(w, \beta)$
11:     Fix $w$ and update architecture parameters $\beta$ by descending $\nabla_\beta(\ell(w, \beta) + \lambda \sum_i \mathrm{Size}(Emb_i))$
12: end for
13: Initialize $ratios[i] = 0 \quad \forall i$
14: for $i = 1, \ldots, c-1$ do
15:     for $l = 0, \ldots, d-1$ do
16:         $ratios[i] += 1$   if $\beta_{i,l} > \beta_i^*$
            ▷ Only save singular values whose importance is larger than threshold
17:     end for
18: end for
19: return $ratios$
```

the residual and directly overwrite the t input tokens as shown in Figure 4(b). Thereby, we can reduce the execution memory to $s \cdot d + t \cdot 5d$. As t is usually much smaller than s, 6× memory reduction can be achieved. In practice, by analytically computing the relation between t and the peak memory usage, we can directly choose t based on the MCU SRAM size. Hence, such scheduling optimization incurs negligible latency overhead on commodity MCUs.

MHA Scheduling. Unlike MLP, the computation of different tokens depends on each other, making it hard to fully tile the MHA computation. We observe the following optimization opportunities: 1) the computation of each head is independent, and 2) the score tensor accounts for the major bottleneck, shown in Figure 1, and its computation can be tiled along token dimension.

Based on the observation, we propose a new MHA scheduling in Figure 5(b). We first tile the computation along the head dimension, i.e., h, and then, further tile the query tensor along the token dimension. This indicates we compute the attention between t tokens and all the s tokens per head each time. It enables us to reduce the memory of the score matrix, breaking the quadratic increase of execution memory into a linear increase with s. Again, we carefully choose t considering both memory constraints and computation parallelism to minimize latency overhead.

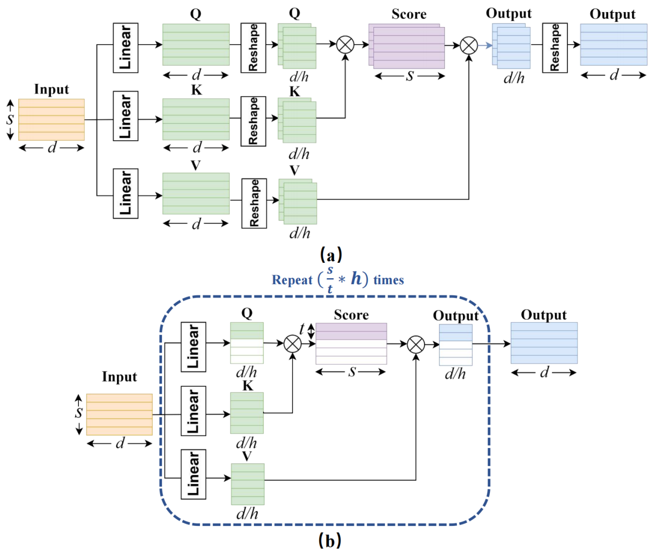


Figure 5: MHA scheduling to reduce the tensor transformation latency and peak memory.

Our proposed MHA scheduling shares similarities with FlashAttention [11], which tiles the key and value matrix. However, FlashAttention requires updating the output matrix repetitively. Either the output matrix has to be stored in high precision, leading to a higher peak execution memory, or quantization and de-quantization operations are needed during each output accumulation step, bringing heavy computation pressure and accuracy loss. Our method is more MCU-friendly and will not face such issues.

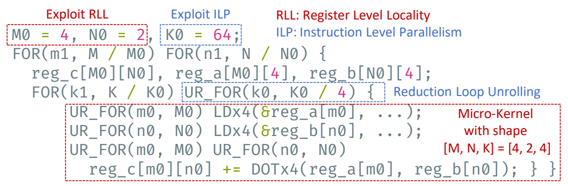
Kernel Design Optimization. The scheduling optimization reduces the peak memory for MLP and MHA. To reduce latency, we also design optimized kernels for each tile of computation.

First, we apply a two-level loop blocking to better fit into the Register-SRAM hierarchy of MCU. Figure 6 illustrates the kernel design for each tile of the matrix multiplication/linear operator. The innermost block is referred to as a micro-kernel, which computes entirely on registers. Properly setting its shape helps exploit register-level locality, thus reducing inference latency. In our experiments, we set the micro-kernel shape to [M, N, K] = [4, 2, 4]. In comparison, CMSIS-NN [20] employs a shape of [1, 2, 4], which exhibits lower locality and decreases performance. Second, we unroll the reduction loop with a factor of 64 to harness the ILP of MCU and better utilize the hardware instruction pipeline. Moreover, by designing the memory access patterns for the linear operators, we fuse all the tensor shape transformation operators and avoid dedicated memory accesses. The kernel design optimization enables to fully leverage the hardware characteristics of MCUs for more efficient BERT processing.

4 EXPERIMENTS

4.1 Experiment Setup

Dataset. We search and evaluate our models on the General Language Understanding Evaluation (GLUE) benchmark [37], which is a collection of text classification tasks. For most evaluations and comparisons, we leverage the Multi-Genre Natural Language Inference Corpus (MNLI) dataset, which is the largest dataset in GLUE.



```
M0 = 4, N0 = 2, K0 = 64;
FOR(m1, M / M0) FOR(n1, N / N0) {
  reg_c[M0][N0], reg_a[M0][4], reg_b[N0][4];
  FOR(k1, K / K0) UR_FOR(k0, K0 / 4) {
     UR_FOR(m0, M0) LDx4(&reg_a[m0], ...);
     UR_FOR(n0, N0) LDx4(&reg_b[n0], ...);
     UR_FOR(m0, M0) UR_FOR(n0, N0)
       reg_c[m0][n0] += DOTx4(reg_a[m0], reg_b[n0]); } }
```

Exploit RLL
Exploit ILP
RLL: Register Level Locality
ILP: Instruction Level Parallelism
Reduction Loop Unrolling
Micro-Kernel
with shape
[M, N, K] = [4, 2, 4]

Figure 6: Matrix multiplication kernel design of MCUBERT. UR_FOR means fully unrolled for-loop. LDx4 means loading 4 consecutive elements from SRAM to the register. DOTx4 means computing dot-product of two 4-element vectors using SIMD instruction (e.g., SMLAD for ARM Cortex-M MCU). The quantization and writing-back operations are omitted here.

Table 3: Accuracy comparison on MNLI. The ratio of the first cluster is not shown as it equals to embedding dimension (Emb stands for embedding, params stands for parameters, and Acc stands for accuracy).

Model	Clustering Cutoffs	Ratios	Emb Params (M)	Total Params (M)	Acc (%)
BERT-tiny	-	-	3.907M	4.300M	70.01%
FWSVD	-	-	0.368M	0.761M	60.83%
Adaptive Emb	1000,4000,10000	32,8,2	0.318M	0.712M	67.01%
MCUBERT-tiny	510,1065, 1915	109, 18, 2	0.231M	0.624M	68.33%
	1218, 2534, 4061	54, 2, 2	0.307M	0.700M	69.18%
BERT-mini	-	-	7.814M	10.960M	74.80%
Adaptive Emb	1000,4000,10000	64, 16, 4	0.648M	3.794M	69.25%
MCUBERT-mini	1014, 3348, 4912	41, 41, 2	0.492M	3.638M	71.89%
	1354, 4301, 6094	41, 42, 2	0.613M	3.759M	72.59%
	1871, 5591, 7788	52, 18, 2	0.779M	3.925M	74.22%

Searching setting. We select lightweight BERT models, i.e., BERT-tiny and BERT-mini for our experiments, and the pre-trained models are adopted from [4, 36]. For NAS, we use AdamW optimizer with a zero weight decay. We use a batch size of 32 for training and set the learning rate to 5×10^{-5}.

Model deployment. We deploy our model on different MCUs, i.e., NUCLEO-F746 with 320KB SRAM and 1MB Flash, NUCLEO-F767 with 512KB SRAM and 2MB Flash as well as NUCLEO-H7A3ZI-Q with 1.4 MB SRAM and 2MB Flash, to measure the latency and the peak memory usage. The batch size is fixed to 1.

4.2 Importance of two-stage NAS

As described in Section 3, we propose a two-stage NAS strategy, searching token clustering in the first stage and approximation ratios in the second stage. To demonstrate the importance of such two-stage formulation, we compare MCUBERT with other baselines based on BERT-tiny and MNLI dataset, as shown in Figure 7, including 1) baseline adaptive embedding; 2) one-stage NAS that searches the token clustering and approximation ratios together (DARTS

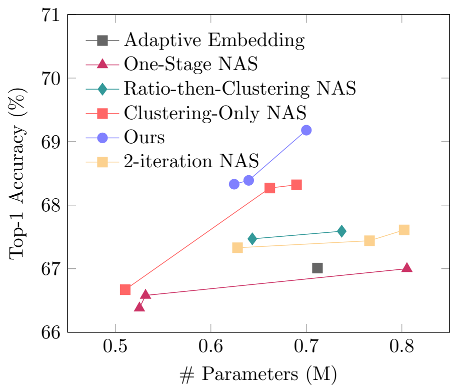


Figure 7: Importance of two-stage NAS.

Table 4: Accuracy comparison on other GLUE datasets (Emb stands for embedding, and params stands for parameters).

Model	Metrics	MRPC	SST2	QQP
BERT-tiny	Emb Params	3.91M	3.91M	3.91M
	Accuracy	74.02%	82.45%	87.16%
Adaptive Emb	Emb Params	0.32M	0.32M	0.32M
	Accuracy	70.34%	81.42%	84.32%
MCUBERT-tiny	Emb Params	0.31M	0.26M	0.32M
	Accuracy	73.77%	82.11%	85.20%
Model	Metrics	MRPC	SST2	QQP
BERT-mini	Emb Params	7.81M	7.81M	7.81M
	Accuracy	78.90%	85.32%	89.25%
Adaptive Emb	Emb Params	0.65M	0.65M	0.65M
	Accuracy	75.49%	83.60%	87.93%
MCUBERT-mini	Emb Params	0.65M	0.49M	0.52M
	Accuracy	77.94%	83.83%	88.12%

[26]); 3) searching approximation ratios first followed by token clustering (denoted as Ratio-then-Clustering NAS); 4) clustering-only NAS that only searches the token clustering; and 5) 2-round NAS that repeats the two-stage NAS for 2 rounds. As shown in Figure 7, our two-stage NAS outperforms all the other strategies and achieves the best Pareto front. One-stage NAS that ideally has the largest search space produces the worst Pareto front. This is because of the large discrepancy among optimal low-rank approximation ratios for different token clustering, which brings convergence difficulties as discussed in Section 3.

4.3 Accuracy Comparison on GLUE

Comparison on MNLI. We compare MCUBERT with the baseline FWSVD [17] and adaptive embedding [1] on the MNLI dataset, as shown in Table 3. For both BERT-tiny and BERT-mini, MCUBERT can simultaneously achieve better accuracy and smaller parameter size compared to the baseline FWSVD and adaptive embedding. Specifically, for BERT-tiny, MCUBERT can achieve 1.4× and 1.6× embedding parameter reduction with 1.3% and 7.5% better accuracy compared to adaptive embedding and FWSVD, respectively. With the same model size, MCUBERT improves the accuracy by 2.2% and 8.4%. For BERT-mini, MCUBERT achieves 3.3% better accuracy with a smaller parameter size compared to adaptive embedding. We also observe for BERT-tiny, MCUBERT tries to assign a higher approximation ratio for the second cluster but assign more tokens to the fourth cluster.

Comparison on Other Datasets. We then compare MCUBERT with adaptive embedding on other GLUE datasets. Note we focus on the MRPC, SST2, and QQP datasets as even the full BERT-tiny and BERT-mini suffers from accuracy issues on COLA and RTE datasets [13]. As shown in Table 4, MCUBERT consistently outperforms adaptive embedding on all three tasks by 3.43%, 0.69%, 0.88% better accuracy for BERT-tiny, respectively. For different datasets, the number of tokens and the approximation ratio of each cluster are different, indicating the necessity of task loss-guided compression.

4.4 Peak Memory and Latency Comparison

Peak memory comparison. We compare the inference latency and peak memory with the baseline CMSIS-NN [20], [5], and FlashAttention [11]. We re-implement [5] and [11] based on the original paper. When re-implement FlashAttention, we need to store the output matrix in high precision, which has been discussed in Section 3.3. As shown in Figure 8, both CMSIS-NN and [5] do not consider the execution memory in the scheduling, leading to a fast memory growth and out-of-memory issue for a long sequence length, i.e., 512. For BERT-tiny, MCUBERT achieves more than 1.9× and 3.5× peak memory reduction compared to the baseline when the sequence lengths are 64 and 512, respectively. This indicates MCUBERT can support 3×, 2×, and 2× longer input sequences compared to CMSIS-NN and [5] on MCUs with 512 KB, 320 KB and 128 KB SRAM, respectively. In FlashAttention, the output matrix needs iterative updates, demanding fp32 data format instead of int8 precision, causing substantial memory usage growth. Moreover, typical FlashAttention implementation keeps the entire query, key, and value matrices in memory, further increasing memory consumption compared to our approach. Compared with FlashAttention, MCUBERT also achieves more than 1.6× and 1.9× peak memory reduction in all sequence lengths for BERT-tiny and BERT-mini, respectively.

Latency comparison. The latency comparison for different sequence lengths is shown in Table 5. MCUBERT achieves around 1.5× and 1.3× latency reduction compared to CMSIS-NN and [5] consistently for different sequence lengths and different MCUs. We visualize the latency breakdown for [5] and MCUBERT. As shown in Figure 10, MCUBERT reduces the latency of both the linear and matrix multiplication operators and fuses all tensor shape transformation operators, demonstrating the high efficiency of our kernel design. Compared with other acceleration strategies such as FlashAttention, MCUBERT still achieves lower latency, although MCUBERT already shows lower peak memory usage than FlashAttention. *With the model and scheduling optimization, MCUBERT enables to deploy BERT-tiny on NUCLEO-F746 with sequence length of 512 for the first time.*

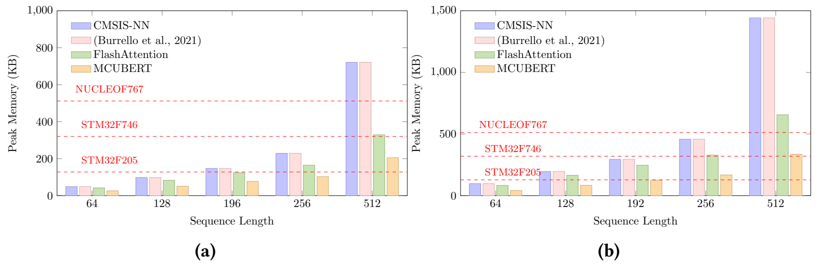


Figure 8: Peak memory comparison for different sequence lengths for (a) BERT-tiny and (b) BERT-mini.

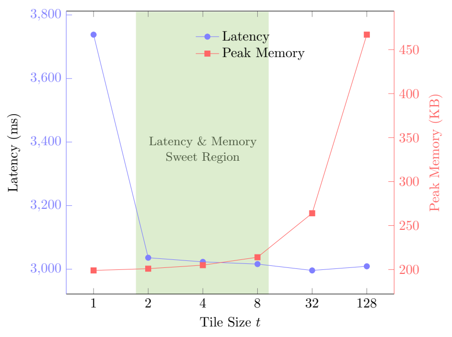


Figure 9: Comparison of latency and memory with different tile sizes t.

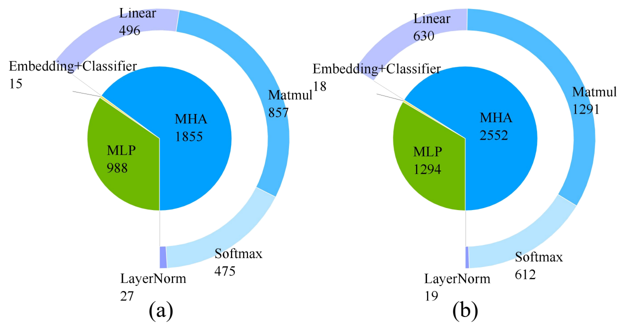


Figure 10: Latency (ms) breakdown for (a) MCUBERT and (b) [5] with 512 sequence length (Matmul stands for matrix multiplication.).

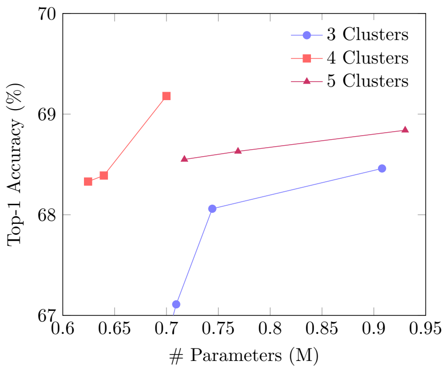


Figure 11: Impact of the number of clusters.

Table 5: Latency (ms) comparison on NUCLEO-F767 (F7) and NUCLEO-H7A3ZI-Q (H7) for different sequence lengths using BERT-tiny (OOM stands for out of memory).

MCU	Methods	64	128	192	256	512
F7	CMSIS-NN	386	838	1355	1939	OOM
	[5]	366	780	1263	1812	OOM
	FlashAttention	273	606	880	1463	4012
	MCUBERT	264	578	942	1354	3591
H7	CMSIS-NN	320	697	1138	1638	4233
	[5]	284	625	1016	1471	3866
	FlashAttention	220	491	818	1205	3297
	MCUBERT	213	469	768	1116	2940

4.5 Ablation Study

Impact of the number of clusters. MCUBERT follows adaptive embedding and empirically fix the number of clusters to 4. We now evaluate its impact on the accuracy-parameter Pareto front. We use the BERT-tiny model and MNLI dataset for comparison. As shown in Figure 11, the Pareto front for $c = 4$ indeed outperforms that for $c = 3$ or $c = 5$. We observe $c = 3$ performs much worse when the model parameter size is small. We hypothesize this is because when $c = 3$, important tokens are forced to have a small dimension, leading to low accuracy.

Impact of fine-grained scheduling. To evaluate the impact of fine-grained scheduling on both latency and memory, we change the number of tokens in each tile, i.e., t. The change in latency and peak memory is plotted in Figure 9. As we can observe, the peak execution memory reduces consistently with the decrease of t while the latency remains roughly the same for $t \leq 2$. The high latency for $t = 1$ is because of hardware under-utilization.

5 CONCLUSION

This work proposes MCUBERT, a network/scheduling co-optimization framework enabling BERT on MCUs. For network optimization, MCUBERT proposes an MCU-aware two-stage NAS algorithm with clustered low-rank approximation for embedding compression. For scheduling optimization, we leverage tiling, in-place computation, and kernel optimization to simultaneously reduce peak memory and latency. MCUBERT overcomes all existing baselines and enables to run the lightweight BERT models on commodity MCUs for the first time.

ACKNOWLEDGMENTS

This work was supported in part by the NSFC (62125401), the 111 Project (B18001), the Theme-Based Research Scheme (TRS) Project T45-701/22-R and General Research Fund (GRF) Project 17203224 of the Research Grants Council (RGC), HKSAR.

REFERENCES

[1] Alexei Baevski and Michael Auli. 2018. Adaptive input representations for neural language modeling. *arXiv preprint arXiv:1809.10853* (2018).

[2] Haoli Bai, Wei Zhang, Lu Hou, Lifeng Shang, Jing Jin, Xin Jiang, Qun Liu, Michael Lyu, and Irwin King. 2020. Binarybert: Pushing the limit of bert quantization. *arXiv preprint arXiv:2012.15701* (2020).

[3] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. 2021. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems* 3 (2021), 517–532.

[4] Prajjwal Bhargava, Aleksandr Drozd, and Anna Rogers. 2021. Generalization in NLI: Ways (Not) To Go Beyond Simple Heuristics. arXiv:2110.01518 [cs.CL]

[5] Alessio Burrello, Moritz Scherer, Marcello Zanghieri, Francesco Conti, and Luca Benini. 2021. A microcontroller is all you need: Enabling transformer execution on low-power iot endnodes. In *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*. IEEE, 1–6.

[6] Daniel Campos, Alexandre Marques, Tuan Nguyen, Mark Kurtz, and ChengXiang Zhai. 2022. Sparse* bert: Sparse models are robust. *arXiv preprint arXiv:2205.12452* (2022).

[7] Patrick Chen, Si Si, Yang Li, Ciprian Chelba, and Cho-Jui Hsieh. 2018. Groupreduce: Block-wise low-rank approximation for neural language model shrinking. *Advances in Neural Information Processing Systems* 31 (2018).

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[9] Yu-An Chung, Yu Zhang, Wei Han, Chung-Cheng Chiu, James Qin, Ruoming Pang, and Yonghui Wu. 2021. W2v-bert: Combining contrastive learning and masked language modeling for self-supervised speech pre-training. In *2021 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE, 244–250.

[10] Baiyun Cui, Yingming Li, Ming Chen, and Zhongfei Zhang. 2019. Fine-tune BERT with sparse self-attention mechanism. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*. 3548–3553.

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

[12] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. 2021. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems* 3 (2021), 800–811.

[13] 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).

[14] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 389–402.

[15] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough. 2019. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers. *Advances in Neural Information Processing Systems* 32 (2019).

[16] Igor Fedorov, Ramon Matas, Hokchhay Tann, Chuteng Zhou, Matthew Mattina, and Paul Whatmough. 2022. UDC: Unified DNAS for Compressible TinyML Models for Neural Processing Units. *Advances in Neural Information Processing Systems* 35 (2022), 18456–18471.

[17] Yen-Chang Hsu, Ting Hua, Sungen Chang, Qian Lou, Yilin Shen, and Hongxia Jin. 2022. Language model compression with weighted low-rank factorization. *arXiv preprint arXiv:2207.00112* (2022).

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

[19] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2019. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351* (2019).

[20] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601* (2018).

[21] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942* (2019).

[22] Yinan Liang, Ziwei Wang, Xiuwei Xu, Yansong Tang, Zhou Jie, and Jiwen Lu. 2023. MCUFormer: Deploying Vision Tranformers on Microcontrollers with Limited Memory. *arXiv preprint arXiv:2310.16898* (2023).

[23] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. 2021. Mcunetv2: Memory-efficient patch-based inference for tiny deep learning. *arXiv preprint arXiv:2110.15352* (2021).

[24] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. 2020. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems* 33 (2020), 11711–11722.

[25] Vasileios Lioutas, Ahmad Rashid, Krtin Kumar, Md Akmal Haidar, and Mehdi Rezagholizadeh. 2019. Distilled embedding: non-linear embedding factorization using knowledge distillation. (2019).

[26] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[27] Shishir G Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph Gonzalez. 2022. Poet: Training neural networks on tiny devices with integrated rematerialization and paging. In *International Conference on Machine Learning*. PMLR, 17573–17583.

[28] Haotong Qin, Yifu Ding, Mingyuan Zhang, Qinghua Yan, Aishan Liu, Qingqing Dang, Ziwei Liu, and Xianglong Liu. 2022. Bibert: Accurate fully binarized bert. *arXiv preprint arXiv:2203.06390* (2022).

[29] 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.

[30] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[31] Manuele Rusci, Alessandro Capotondi, and Luca Benini. 2020. Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers. *Proceedings of Machine Learning and Systems* 2 (2020), 326–335.

[32] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).

[33] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2020. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8815–8821.

[34] Han Shi, Jiahui Gao, Xiaozhe Ren, Hang Xu, Xiaodan Liang, Zhenguo Li, and James Tin-Yau Kwok. 2021. Sparsebert: Rethinking the importance analysis in self-attention. In *International Conference on Machine Learning*. PMLR, 9547–9557.

[35] Joonbo Shin, Yoonhyung Lee, and Kyomin Jung. 2019. Effective sentence scoring method using BERT for speech recognition. In *Asian Conference on Machine Learning*. PMLR, 1081–1093.

[36] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Well-Read Students Learn Better: The Impact of Student Initialization on Knowledge Distillation. *CoRR* abs/1908.08962 (2019). arXiv:1908.08962 http://arxiv.org/abs/1908.08962

[37] 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).

[38] Hu Xu, Bing Liu, Lei Shu, and Philip S Yu. 2019. BERT post-training for review reading comprehension and aspect-based sentiment analysis. *arXiv preprint arXiv:1904.02232* (2019).

[39] Jin Xu, Xu Tan, Renqian Luo, Kaitao Song, Jian Li, Tao Qin, and Tie-Yan Liu. 2021. NAS-BERT: task-agnostic and adaptive-size BERT compression with neural architecture search. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1933–1943.

[40] Jiacheng Yang, Mingxuan Wang, Hao Zhou, Chengqi Zhao, Weinan Zhang, Yong Yu, and Lei Li. 2020. Towards making the most of bert in neural machine translation. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 9378–9385.

[41] Ofir Zafrir, Ariel Larey, Guy Boudoukh, Haihao Shen, and Moshe Wasserblat. 2021. Prune once for all: Sparse pre-trained language models. *arXiv preprint arXiv:2111.05754* (2021).

[42] Yujia Zhai, Chengquan Jiang, Leyuan Wang, Xiaoying Jia, Shang Zhang, Zizhong Chen, Xin Liu, and Yibo Zhu. 2023. ByteTransformer: A high-performance transformer boosted for variable-length inputs. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 344–355.

[43] Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. 2020. Ternarybert: Distillation-aware ultra-low bit bert. *arXiv preprint arXiv:2009.12812* (2020).

[44] Xiaofan Zhang, Zongwei Zhou, Deming Chen, and Yu Emma Wang. 2022. AutoDistill: An end-to-end framework to explore and distill hardware-efficient language models. *arXiv preprint arXiv:2201.08539* (2022).

[45] Jinhua Zhu, Yingce Xia, Lijun Wu, Di He, Tao Qin, Wengang Zhou, Houqiang Li, and Tie-Yan Liu. 2020. Incorporating bert into neural machine translation. *arXiv preprint arXiv:2002.06823* (2020).