A Compression-Compilation Framework for On-mobile Real-time BERT Applications

Wei Niu[1*], **Zhenglun Kong**[2*], **Geng Yuan**[2], **Weiwen Jiang**[3], **Jiexiong Guan**[1], **Caiwen Ding**[4], **Pu Zhao**[2], **Sijia Liu**[5], **Bin Ren**[1] and **Yanzhi Wang**[2]
[1]Computer Science, College of William and Mary, USA
[2]Electrical & Computer Engineering, Northeastern University, Boston, USA
[3]Computer Science & Engineering, University of Notre Dame, USA
[4]Computer Science & Engineering, University of Connecticut, USA
[5]MIT-IBM Watson AI Lab, IBM Research
{wniu,jguan, bren}@email.wm.edu, {kong.zhe,yuan.geng,zhao.pu, yanz.wang}@northeastern.edu, wjiang2@nd.edu, caiwen.ding@uconn.edu, sijia.liu@ibm.com

Abstract

Transformer-based deep learning models have increasingly demonstrated high accuracy on many natural language processing (NLP) tasks. In this paper, we propose a compression-compilation co-design framework that can guarantee the identified model to meet both resource and real-time specifications of mobile devices. Our framework applies a compiler-aware neural architecture optimization method (CANAO), which can generate the optimal compressed model that balances both accuracy and latency. We are able to achieve up to 7.8× speedup compared with TensorFlow-Lite with only minor accuracy loss. We present two types of BERT applications on mobile devices: Question Answering (QA) and Text Generation. Both can be executed in real-time with latency as low as 45ms. Videos for demonstrating the framework can be found on https://www.youtube.com/watch?v=_WIRvK_2PZI

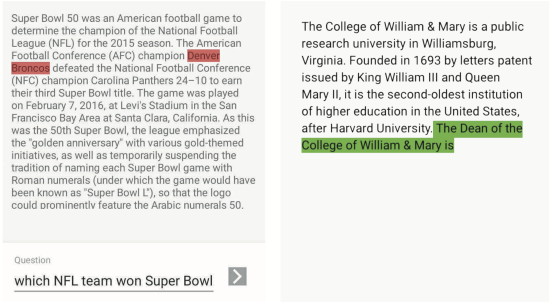

Figure 1: Screenshot of the real-time BERT application on smartphone. Left: Question Answering task. Type a random question that is related to the paragraph, it will automatically highlight the answer in the test. Right: Text Generation task. Given a starting sentence, it can automatically generate new sentences by word.

1 Introduction

Pre-trained large-scale language models such as BERT [Devlin *et al.*, 2019], XLNet [Yang *et al.*, 2019], RoBERTa [Liu *et al.*, 2019] have substantially advanced the state-of-the-art across a wide spectrum of NLP tasks. With the increasing popularity of mobile AI applications and the concerns of information security and privacy, it is desirable to deploy these well-trained models on edge devices, and furthermore, to meet real-time requirements. However, these models often consist of hundreds (or even thousands) of computation layers and hundreds of millions of parameters. Therefore, how to accommodate the large and extremely deep models, such as BERT to edge device becomes an imminent problem.

There have been some efforts to compress the BERT model while maintaining the accuracy for downstream NLP tasks. MobileBERT [Sun *et al.*, 2020] is able to reduce the memory requirement, but there is still a considerable execution overhead due to a large number of computation units, thus leading to high inference latency. Moreover, the large number of model layers also brings challenges in compiling models to mobile devices. To the best of our knowledge, only TensorFlow-Lite (TFLite) [Abadi *et al.*, 2015] supports deploying BERT models on mobile CPU (not on mobile GPU), while no other frameworks can even support BERT models on mobile CPU.

In this paper, we propose a compression-compilation co-design framework to optimize the structures of BERT variants for mobile devices. This is the first framework that involves compiler optimizations in the architecture search loop, aiming to co-optimize the model accuracy and computation resource usage. We also propose a highly effective layer fusion method to reduce intermediate results to achieve lower latency on both mobile CPU and GPU. Our framework outperforms the state-of-the-art framework, TFLite, by up to 7.8× speedup. Thus achieving the least latency while executing on mobile devices. We will release our model and framework.

*These authors contributed equally

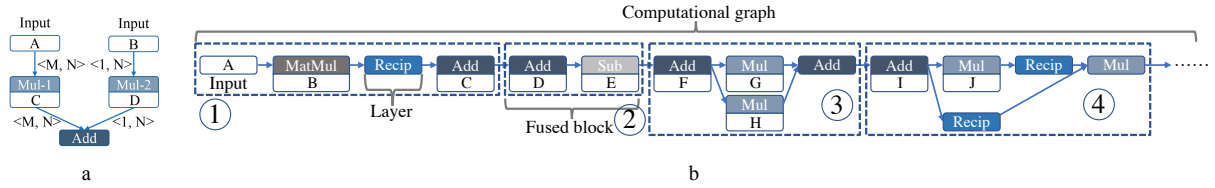

Figure 2: a. Fusion example demonstrated in Figure 4. b. Sample fusion candidates for a computational graph section with an input (marked with A). Each layer has an input either from the previous layer/layers or from its weights, marked with other alphabets. Each number (from 1 to 4) denotes a fusion candidate (or fused block) based on mathematical properties.

2 Framework Design

There are two processes in CANAO: *training* and *compiler code generation* (as shown in Figure 3). The training process includes a controller and a trainer. The controller predicts/generates the model hyperparameters (i.e., network architecture); the trainer trains the predicted model and (quickly) evaluates its accuracy by fine-tuning the model to downstream tasks. The compiler code generation process takes the predicted model and returns execution information (e.g. number of fused layers, latency, CPU/GPU utilization). The execution information together with the model accuracy from the training process will be feedback to the controller to improve the prediction of neural architectures. After the compiler-aware NAS, the generated codes by our optimized compiler will be deployed for mobile CPU/GPU executions.

For the training process, the controller generates the architectural hyperparameters of neural networks. This includes two phases: 1) The determination of the number of transformer blocks; 2) The optimization of size for each layer. We find that layer number affects the accuracy the most for BERT related models, thus it should be the first thing we determine when searching the optimized model architecture. Then we optimize the layer size by considering both inference latency and model accuracy, which are set as reward signals to feedback to the controller. The controller serves to find the optimal architecture by maximizing the expected reward.

The compiler code generation process includes three steps: 1) Generate a computational graph from the controller-generated model and apply multiple optimizations on this graph. 2) Employ a novel compiler-based layer fusion optimization to further improve execution performance. This plays a key role in achieving better hardware efficiency. 3) Employ code generation and optimization to generate and further optimize the inference code. The generated inference code is tested on mobile devices. According to the feedback from the device side, the controller makes a better tradeoff between model accuracy and latency.

2.1 Controller Architecture Search

Our search space includes the number of layers, hidden layer size, and intermediate embedding size of the feedforward layers. We apply the recurrent neural network for searching the model architecture in the Controller. The recurrent network can be trained with a policy gradient method to maximize the expected reward of the sampled architectures. The accuracy and latency are used as the reward signal to feedback to the controller, which is trained by using the reinforcement learning method to explore the architecture. Our framework can search for a desirable model that achieves a good balance between accuracy and latency, preventing from searching the architecture manually.

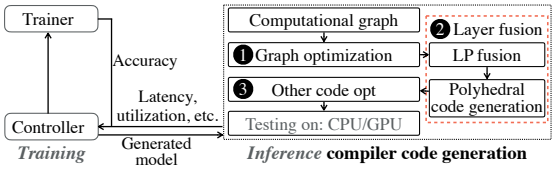

Figure 3: Overview of compiler-aware neural architecture optimization framework.

2.2 Compiler Code Generation

This section introduces our compiler optimizations that optimize the latency reward for the feedback. There are two phases: Lightweight Polynomial-based Layer Fusion (LP-Fusion) and Polyhedral-based Code Generation.

LP-Fusion We identify all fusion candidates in a model based on two kinds of properties in the polynomial calculation: *computation laws* (i.e., associative, commutative, and distributive) and *data access patterns*. Fig. 2b shows four fusion candidates (or fused blocks) for a computational graph. Layer fusion reduces not only the memory consumption of intermediate results, but also the number of operators. Take Fig. 2b③ for example, without layer fusion, the computation function is defined as:$(\star + F) \odot G + (\star + F) \odot H$. The layer and computation count numbers are 4 and 5, respectively. After fusion, the computation function is simplified as:$(\star + F) \odot (G + H)$. Where layer and computation count numbers become 1 and 3, respectively. This process can significantly reduce the operator number and computation overhead. Compared with prior work on loop fusion [Ashari *et al.*, 2015; Bezanson *et al.*, 2017; Boehm *et al.*, 2018], the novelty of this approach is that we exploit a restricted domain of DNN execution. Thus, we can enable more aggressive optimizations without very expensive exploration.

Polyhedral-based Code Generation LP-Fusion supports grouping multiple layers with varied output shapes, i.e., in the code-level, the nested loop structures of these layers may be different. Traditional compilers cannot support this kind of loop fusion well, mainly due to the complexity of such loop analysis. As shown in Fig. 2a and Figure 4, there are three operators: `Mul-1`, `Mul-2`, and `Add`. `Mul-1` and `Mul-2` take

Framework Device	#FLOPs	TFLite CPU	CANAO (without layer fusion) CPU	Speedup	 GPU	Speedup	CANAO (with layer fusion) CPU	Speedup	 GPU	Speedup
DistilBERT with NAS	10.9G	188ms	157ms	1.2×	237ms	0.8×	**105ms**	**1.8**×	**86ms**	**2.2**×
$\text{BERT}_{\text{BASE}}$ with NAS	21.8G	352ms	276ms	1.3×	412ms	0.9×	**196ms**	**1.8**×	**147ms**	**2.4**×
CANAOBERT with NAS	4.6G	98ms	89ms	1.1×	152ms	0.6×	**49ms**	**2.0**×	**45ms**	**2.2**×

Table 1: Inference latency comparison of CANAO framework and TFLite on mobile CPU and GPU. All models are generated with English Wikipedia dataset. TFLite does not support BERT on mobile GPU.

```
func mul1: T *in0, T *in1, int row, int col, T *out
  for i = 0 to i < row
    for j = 0 to j < col
      let idx = i * col + j
       out[idx] = in0[idx] * in1[idx]
func mul2: T *in0, T *in1, int col, T *out
  for j = 0 to j < col
    out[j] = in0[j] * in1[j]
func fuse_add: T* in0, T* in1, T* in2, T* in3, int row, int col, T* out
  for i = 0 to i < row
    for j = 0 to j < col
      let idx = i * col + j
      out[idx] = in0[idx] * in1[idx] + in2[j] * in3[j]
func fuse_add': T* in0, T* in1, T* in2, T* in3, int row, int col, T* out
  for j = 0 to j < col
    let temp = in2[j] * in3[j]
    for i = 0 to i < row
      let idx = i * col + j
      out[idx] = in0[idx] * in1[idx] + temp
```

Figure 4: An example of loop fusion.

matrices A and B as their input, respectively. `Add` takes the output of `Mul-1` and `Mul-2` as input to generate the result. The shape of matrix A is M $\times$ N while the shape of B is 1 $\times$ N. We have two options to perform loop fusion:`fuse_add`, and `fuse_add'`. In case `fuse_add`, `in2` and `in3` are single dimensional arrays, so `in2[j] * in3[j]` incurs redundant computation for each outer-loop iteration (except the first one). The case `fuse_add'` resolves this redundant computation with a proper loop permutation; however, it degrades the data locality because memory access for both matrices `in0` and `in1` becomes column-major, inconsistent with their memory storage. To address this complexity, our compiler extends the polyhedral analysis model [Wilde, 1993] to generate both versions and employs auto-tuning to dynamically select the optimal version. Moreover, our compiler also employs an extended polyhedral analysis model to analyze the loop structure and data dependency, transform indices, and generate optimal fused code for other loop fusion cases.

3 Experiments and Demonstrations

3.1 Training and Evaluation Setup

Our models are trained on a server with 16× NVIDIA Tesla V100 (Volta) GPUs. We use English Wikipedia [Devlin *et al.*, 2019] and BooksCorpus [Zhu *et al.*, 2015] to train the models and finetune on GLUE benchmark [Wang *et al.*, 2018]: MNLI [Williams *et al.*, 2018], SST-2 [Socher *et al.*, 2013], MRPC [Dolan and Brockett, 2005], STS-B [Cer *et al.*, 2017], RTE [Wang *et al.*, 2018], and CoLA [Warstadt *et al.*, 2019]. The sequence length is 128. We evaluate our framework on a Samsung Galaxy S20 cell phone with Qualcomm Snapdragon 865. For each model, we run our framework and TFLite 100 times with 8 threads on CPU and all pipelines on GPU.

Model	MNLI-m/mm	SST-2	MRPC	STS-B	RTE	CoLA
$\text{BERT}_{\text{BASE}}$	84.6/83.4	93.5	88.9	85.8	66.4	52.1
DistilBERT	81.5/81.0	92	85.0	-	65.5	51.3
MobileBERT	83.3/82.6	92.8	88.8	84.4	66.2	50.5
CANAOBERT	82.9/82.1	92.6	88.4	83.5	65.6	49.2

Table 2: Evaluation accuracy results on GLUE benchmark. All models are optimized with layer fusion and code generation (i.e., they already run faster than their TFLite implementation) with a fixed sequence length of 128.

3.2 Demonstration on Mobile

Figure 1 shows the interface of our real-time BERT application. Figure 1 left is the Question Answering task. Type a random question that is related to the paragraph, it will automatically highlight the answer in the test. Figure 1 right is the Text Generation task. Given a starting sentence, it can automatically generate new sentences by word.

3.3 Evaluation Results

We compare the accuracy and latency of four models: $\text{BERT}_{\text{BASE}}$ [Devlin *et al.*, 2019], MobileBERT [Sun *et al.*, 2020], DistilBERT [Sanh *et al.*, 2019], and CANAOBERT. Table 2 shows the accuracy and latency results. We manage to significantly reduce latency compared to $\text{BERT}_{\text{BASE}}$, DistilBERT, and MobileBERT on both CPU and GPU. Compared with $\text{BERT}_{\text{BASE}}$, our model is 5.2× faster on CPU and 4.1× faster on GPU with 0.5-2% accuracy loss. Compared with MobileBERT, our model is 1.49 × faster on CPU and 1.53× faster on GPU with only 0.4-1% accuracy decrease.

3.4 Effectiveness of Compiler Optimizations

We compare with a state-of-the-art framework, TFlite. Table 1 shows inference latency comparison results. TFLite only supports mobile CPU execution, and other frameworks do not support BERT models on mobile devices. And GPU performance is unusually slower than CPU (only 0.6× speedup for CANAOBERT over TFLite on CPU). The fully optimized framework can achieve up to 2.0× speedup on CPU, and 2.4× on GPU, over TFLite's CPU execution. Notably, comparing to $\text{BERT}_{\text{BASE}}$ on TFLite (352ms on CPU), our overall model and framework (45ms on GPU) can achieve up to 7.8× speedup.

4 Conclusion

We introduced a novel compression-compilation co-design framework to optimize the structures of BERT variants for mobile devices. We presented two BERT applications on mobile devices: Question Answering and Text Generation. Both can be executed in real-time with latency as low as 45ms.

References

[Abadi *et al.*, 2015] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[Ashari *et al.*, 2015] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P Sadayappan. On optimizing machine learning workloads via kernel fusion. *ACM SIGPLAN Notices*, 50(8):173–182, 2015.

[Bezanson *et al.*, 2017] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[Boehm *et al.*, 2018] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Alexandre V Evfimievski, and Prithviraj Sen. On optimizing operator fusion plans for large-scale machine learning in systemml. *arXiv preprint arXiv:1801.00829*, 2018.

[Cer *et al.*, 2017] Daniel Cer, Mona Diab, Eneko Agirre, Inigo Lopez-Gazpio, and Lucia Specia. Semeval-2017 task 1: Semantic textual similarity multilingual and crosslingual focused evaluation. *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, 2017.

[Devlin *et al.*, 2019] J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, 2019.

[Dolan and Brockett, 2005] Bill Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Third International Workshop on Paraphrasing (IWP2005)*. Asia Federation of Natural Language Processing, January 2005.

[Liu *et al.*, 2019] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, M. Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *ArXiv*, abs/1907.11692, 2019.

[Sanh *et al.*, 2019] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *ArXiv*, abs/1910.01108, 2019.

[Socher *et al.*, 2013] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA, October 2013. Association for Computational Linguistics.

[Sun *et al.*, 2020] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020.

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

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

[Wilde, 1993] Doran K. Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA, December 1993.

[Williams *et al.*, 2018] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122. Association for Computational Linguistics, 2018.

[Yang *et al.*, 2019] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[Zhu *et al.*, 2015] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 19–27, 2015.