2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)

Efficient 1D Grouped Convolution For PyTorch A Case Study: Fast On-Device Fine-Tuning For SqueezeBERT

S. H. Mozafari, J. J. Clark, W. J. Gross, and B. H. Meyer
Department of Electrical and Computer Engineering, McGill University, Montréal, QC, Canada
{seyyed.mozafari, james.clark1, warren.gross, brett.meyer}@mcgill.ca

***Abstract*—Grouped convolution has been observed to be an effective approximation for convolution in many DNN applications. For example, SqueezeBERT, which is a light and fast BERT language processing model, utilizes 1D grouped convolutions. Though SqueezeBERT is well-optimized for inference on edge devices, it suffers from poor memory management during fine-tuning (training). This results in longer fine-tuning time on resource-limited GPUs compared to the original BERT model, BERT-base, despite being specifically designed for edge devices. We study this behavior and show that this poor memory management originates from the use of 1D grouped convolutions in SqueezeBERT. We re-implement 1D grouped convolutions using fully-connected layers, addressing the poor memory allocation and data locality of 1D grouped convolutions. We show that our method is well-suited for edge devices with limited memory; further, it has a negligible effect on inference speed. When utilizing our method, we observe a 42% reduction in fine-tuning time for SqueezeBERT on edge devices.**

I. Introduction

Due to the resource constraints of embedded edge devices such as smartphones, the deployment of deep neural networks (DNNs), like the BERT natural language processing (NLP) model [1], became difficult. However, many of them do not consider the hardware implications (such as latency and energy consumption). For example, BERT-base consumes a staggering 432 MB of memory, while a single inference takes more than 1.7s on a smartphone (pixel 3) [2]. Therefore, compression techniques have been introduced to address this problem.

Convolution is the main operation in many DNNs, and it is very computationally expensive [2], [3], [4], [5]. Many compression techniques revolve around replacing the standard convolutions in a neural network with grouped convolutions [2], [6], [7], [8], [9]. For example, SqueezeBERT [2] utilizes 1D grouped convolutions to compress BERT-base [1]. This allows for substantial savings in computation, which translates to faster inference, with a minimal loss of accuracy. More specifically, SqueezeBERT operates inference with 4.3x faster on edge devices, and it has less than half of the parameters of BERT-base. SqueezeBERT gains this outstanding efficiency using a trick: approximating most of the fully-connected layers in BERT-base encoders with 1D grouped convolutions. Note that we study SqueezeBERT since it easily beats other state of the art embedded BERT architectures (such as MobileBERT [4], ALBERT [10], etc.) in terms of inference speed and/or accuracy on mobile devices (for more information refer to [2]).

However, when evaluating the implementation of SqueezeBERT for **fine-tuning (training)** using PyTorch [11], we observe that the measured training time is far from the expected one. **As shown in Figure 1, we observe that while SqueezeBERT fine-tunes (trains) more quickly than BERT-base on a server-class Tesla K80 (blue lines and the right axis), SqueezeBERT fine-tunes substantially more slowly on the memory-constrained Jetson TX2 (red lines and the left axis).** Note that the number of trainable parameters in SqueezeBERT is less than half of the parameters of BERT-base. We will show that this slowdown in fine-tuning is 1) task agnostic, and 2) results from frequent GPU stalls due to the memory accesses pattern of 1D group convolution, specifically for big batch sizes. Note that fine-tuning must be performed with big batch sizes (the batch size determines the speed and stability of the learning process [12]). Therefore, reducing batch size to better manage on-device memory is not a solution.

On-device fine-tuning is a necessity for NLP models including BERT. Even though cloud computing has allowed complex networks, such as BERT, to be deployed at large scale [14], cloud computing has many drawbacks, including privacy concerns, [15] network latency, and considerable energy overhead for edge devices due to the cost of communication. Simultaneously, edge devices such as smartphones are increasingly capable of compute-heavy tasks. However, the statically trained model in the cloud cannot adapt dynamically in an ever changing user environment and may result in low accuracy for new inputs. Therefore, much research [16], [17] is being done to move traditionally cloud-based systems to the edge. Model designers must therefore take the latency of on-device fine-tuning into account.

We introduce a method that breaks 1D grouped convolutions into small fully-connected layers (non-iterative matrix multiplications) that 1) utilize cache more efficiently, and 2) reduce the memory needed for big batch sizes by allocating memory for activations and training states, group by group. Then, we evaluate the effectiveness of our introduced 1D grouped implementation on SqueezeBERT fine-tuning. Our implementation results in a 42% fine-tuning time reduction for SqueezeBERT on memory- and cache-constrained embedded

DOI 10.1109/ASAP57973.2023.00024

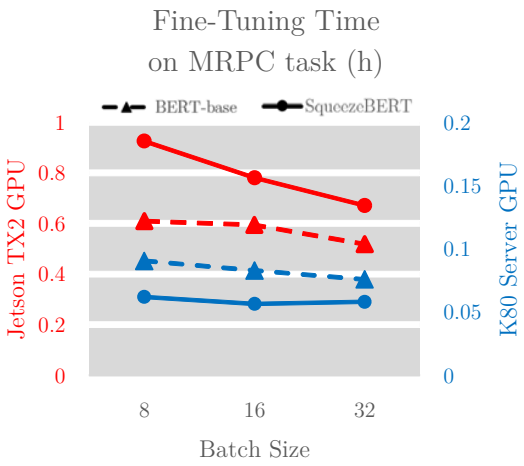


Fig. 1: Fine tuning time (h) vs. batch size for MRPC downstream task [13] on the K80 Tesla server GPU (blue and the right axis), and the embedded Jetson TX2 GPU (red and the left axis).

GPUs, such as the Jetson TX2, while negligibly affecting SqueezeBERT's inference latency, and not affecting its accuracy at all. Our method achieves this by utilizing better 1) memory management, and 2) cache usage, compared to the original SqueezeBERT.

II. Background

In this section, we explain the architecture of BERT and how SqueezeBERT utilizes 1D grouped convolution to optimize BERT. Afterward, we show the 1D grouped convolution memory management pattern and its difficulties in SqueezeBERT as a case study.

A. BERT Architecture

As illustrated in Fig. 2a, BERT models consist of three stages: embedding, encoding, and a task specific classifier, stacked on top of each other [1]. In BERT, the embedding stage converts pre-processed words (represented as integer-valued tokens) into learned feature-vectors of floating-point numbers. Then, these feature vectors are fed into a series of self-attention and other layers (called encoders). In the end a classifier produces the network's final output. As illustrated in Fig. 2a, encoders in BERT models are composed of a stack of different layers. Each encoder has two main sub-layers: the first is a multi-head self-attention unit (which is implemented using fully connected layers), and the second is a simple, position-wise fully-connected feed-forward network. For more details on BERT model architecture, the interested reader is referred to [18].

B. BERT's Bottleneck

Fully-connected (FC) layers, used in multi-head attention and feed-forward layers of BERT, are the latency bottlenecks for this model. Note that FCs are used in multi-head attention and feed-forward layers (refer to Fig. 2a). Profiling the model in Pytorch demonstrates that FC layers, in different BERT's sub-layers, account for more than 97% of the FLOPs and over 80%, and 62% of the inference latency in BERT-base on an Intel Xeon E5-1620 CPU, and K80 GPU, respectively. Our observation is aligned with the results from [2], [19].

C. SqueezeBERT

To improve inference latency, SqueezeBERT [2] utilizes convolutions, and particularly 1D grouped convolutions, to substitute for self-attention and feed-forward layers in encoders. This reduces the complexity of these blocks (reducing the model's trainable parameters by half), and consequently, increases the inference speed of BERT, while maintaining competitive accuracy compared to original BERT implementation (BERT-base).

SqueezeBERT implements the FC layers in its self-attention and feed-forward layers using 1D convolution. Note that for a kernel size of one, convolution acts as a FC layer when the input is the same size of the filter(s) [2]. When the filter size is equivalent to the input size (C_{in}), and there is no padding, the filter cannot convolve over the 1D input array. Therefore, convolution becomes equivalent to a single array of *multiplication and add* units. Moreover, when we have C_{out} filters, C_{out} outputs are generated. In this case, 1D convolution acts as a $C_{in} \rightarrow C_{out}$ fully-connected layer [2].

SqueezeBERT approximates 1D convolutions using 1D grouped convolutions. In a grouped convolution with g groups, filters and input arrays are broken into g groups of C_{in}/g. Then, convolution happens with respect to the groups. This process is shown in Fig. 2b, and it reduces the number of trainable parameters by $1/g$. In other words, each of the filter groups convolves with only $1/g$ of the channel (Cin), resulting in parameter reduction by $1/g$. SqueezeBERT takes advantage of this in both BERT attention heads and feed-forward units. This complexity reduction results in $4.3\times$ inference time improvement in SqueezeBERT with a slight accuracy drop [2].

III. Proposed Methodology

We did an experiment and calculated the unused memory (i.e., accessed but not used) in 1D grouped convolutions utilized in SqueezeBERT. We observed that they constitute more than 23% of the fine-tuning memory for a batch size of 32, while SqueezeBERT runs out of on-chip memory for the Jetson TX2. We propose to replace 1D group convolutions with fully-connected layers. This saves the unused memory in 1D grouped convolutions. Using our method, we reduce the memory utilization of SqueezeBERT considerably, and therefore, prevent off-chip memory accesses, and increase the speed of fine-tuning considerably, while we do not affect accuracy.

Also, this savings improves cache efficiency, where 1D grouped convolution filter weights should be stored before being processed. Our method prevents the caching of unused

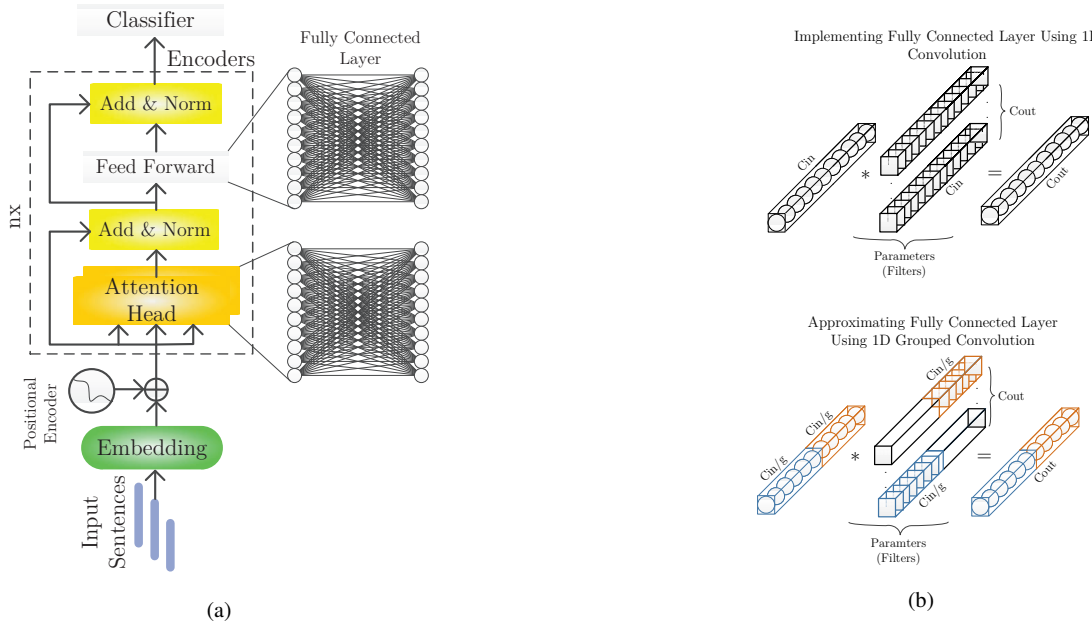


Fig. 2: (a) BERT architecture, and (b) how SqueezeBERT implements FC layers using 1D convolutions and then optimizes them using 1D grouped convolution.

data. Therefore, cache can be utilized more efficiently, which results in faster fine-tuning of SqueezeBERT as an example.

A. Cache and Memory Management in SqueezeBERT Fine-tuning

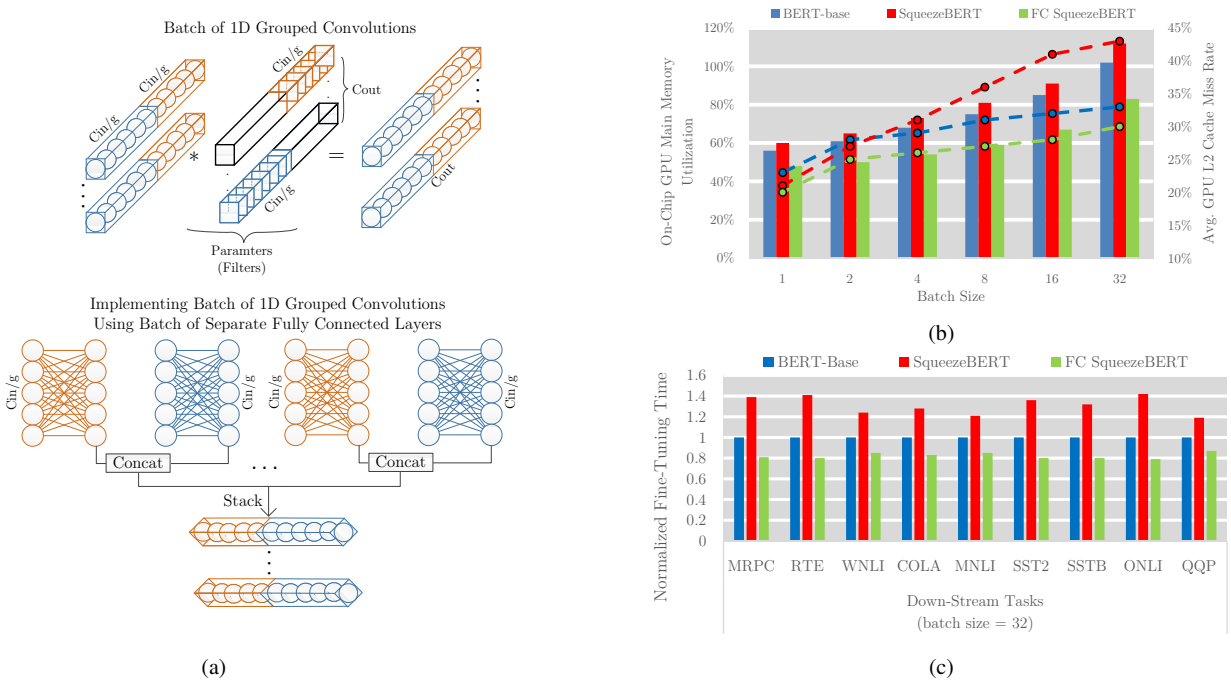
While SqueezeBERT takes advantage of grouped convolution to reduce the number of trainable parameters, implementing FCs using 1D grouped convolutions negatively affects the speed of fine-tuning specifically on cache- and memory-constrained hardware such as embedded GPUs. This happens due to two reasons: 1) the lack of cache data locality in 1D grouped convolution, and 2) inefficient main memory allocation in 1D grouped convolution. The former causes sub-optimal cache performance, specifically for large batch sizes where parameters may need to be reloaded from higher levels of cache or even RAM, while the latter causes inevitable off-chip memory accesses for the GPU.

1) Grouped Convolution Lacks Data Locality: There is no data locality in 1D grouped convolution when the input and filter(s) have the same size. Low-level GPU convolution libraries try to keep all parameters (filters) in the fastest memory (e.g., L2 and L1 caches) to utilize the existing data locality in the convolution algorithm, and take advantage of updating filters concurrently [20]. However, this cache management regime results in a considerable slowdown during training, specifically for 1D grouped convolution with a filter size equivalent to the input size, as in SqueezeBERT. This is the case since for this type of convolution input arrays cannot convolve over filters. To have a better understanding, refer to Fig. 2b. When input has the same size of a filter, there is no space for the filter to convolve over input. In other words, there is no data reuse in this sort of convolution.

To understand the effect of this poor data locality, we conduct some experiments with SqueezeBERT on the Jetson TX2 using PyTorch libraries. In this regard, we measure the L2 cache misses when fine-tuning SqueezeBERT over different batch sizes for the MRPC, a downstream task which is used for fine-tuning BERT models [13]. We measure cache misses by tracking the *L2-write-sector-misses* performance counter (for more details refer to Section IV). As shown in Fig. 3b (blue and red dashed lines, and the right axis), the L2 cache miss rate for the fine-tuning of SqueezeBERT is consistently worse than BERT-base everywhere except for small batch sizes. This considerably higher cache miss rate is one of the factors that slows down SqueezeBERT during fine-tuning.

2) Off-chip Memory Accesses: When implementing grouped convolution, low-level CUDA libraries allocate memory for entire filter arrays even though the filter weights are located just in part of the filter arrays [21]. This poor memory management, while not important for inference or fine-tuning with small batches, results in a shortage of on-chip memory for big batch sizes. Note that we cannot fine-tune models with small batch sizes. This is the case since the batch size determines the speed and stability of the learning process [12].

As illustrated in Fig. 3b (blue and red bars, and the left axis), the memory allocated for fine-tuning SqueezeBERT is more than that for BERT-base, even though it has half the trainable parameters. This is the case since SqueezeBERT utilizes 1D convolutions bigger than the FC layers in BERT-base, and consequently reduces the number of trainable parameters by



(a) (b) (c)

Fig. 3: (a) How to implement 1D grouped convlution using small separate FCs, (b) the main memory utilization and the average L2 cache miss rate of BERT-base and SqueezeBERT over the MRPC task, and (c) normalized fine-tuning time for down stream tasks for different BERT implementations for batch size = 32.

utilizing 1D grouped convolutions. However, as mentioned before, CUDA libraries cannot recover the unused/never accessed memory in 1D grouped convolutions [21]. Therefore, utilizing 1D grouped convolution does not result in any memory saving, specifically in the GPU and CUDA implementation.

As shown in Fig. 3b, this high memory usage for big batch sizes ($\geq$ 32) results in an on-chip memory shortage, which causes off-chip memory accesses and a huge slowdown in training. **Note that memory capacity in embedded systems is very restricted and does not scale like server systems. For example, the next generation of the Jetson (Xavier [22]), which was released three years after the Jetson TX2, does not provide additional on-chip memory.** Therefore, managing memory, specifically for memory-hungry on-device fine-tuning, is crucial.

B. FC Grouped Convolution

Our method implements 1D grouped convolution using independent matrix multiplications (or FC layers). We call our method fully connected (FC) grouped convolution. We utilize it in SqueezeBERT implementation and show that it substantially reduces on-device SqueezeBERT fine-tuning time, while maintaining model accuracy (we call this new implementation of SqueezeBERT, FC SqueezeBERT). As shown in Fig. 3a, we break the arrays for an input batch and parameters (filters) into g groups. (This procedure is shown using blue and orange colors.) Then, we utilize g independent, concurrent FC layers to perform 1D convolution on each input and parameter array. (This process translates to multiplying an input and a filter from a group.) Afterward, we concatenate the outputs for g FC layers to make the corresponding 1D grouped convolution output array. We repeat this process for each input array in a batch. For example, if we use 1D grouped convolution ($Cin \rightarrow Cout$) with two groups, and have two input arrays in each batch, we utilize four FC layers of size $\frac{1}{2}Cin \rightarrow \frac{1}{2}Cout$ to generate the corresponding two output arrays.

As we observed, FC SqueezeBERT reduces memory in 1D grouped convolutions by saving unused memory, and therefore, it reduces off-chip memory accesses. Also, FC SqueezeBERT prevents the cache from being occupied by unused parts of filters in 1D grouped convolutions (refer to Section III-A1). As a result it improves cache efficiency.

IV. Experimental Setup

A. Fine-tuning Data

We utilize the General Language Understanding Evaluation (GLUE) [13] benchmarks to evaluate BERT-base, SqueezeBERT, FC SqueezeBERT. GLUE, which consists of nine tasks, has become the standard evaluation benchmark for NLP research. We fine-tune each task for 3 epochs. We set their max sequence length to 128, and learning rate to 10^{-5}. We train models with different batch sizes, and training always happens on one device.

B. Hardware Platforms

To evaluate the efficiency of FC grouped convolution, we evaluate the speed of fine-tuning of each model (i.e., BERT-base, SqueezeBERT, and FC SqueezeBERT) on GLUE tasks, we utilize a server system equipped with a server-class NVIDIA Tesla K80, and a Jetson TX2 embedded development board, which has an NVIDIA Pascal GPU. The Tesla GPU is equipped with 12GB GDDR5 on-chip memory, and 2496 CUDA cores, while the embedded GPU is equipped with 8GB LPDDR memory (which is shared with its CPU), 512-KB L2 cache, and 256 CUDA cores. Also, to debug our code and extract low-level performance counter info we use NVIDIA system profiler tools, and nvprof. Moreover, to observe the efficiency of our method on embedded CPUs, we utilize the CPU of Hikey 970 board [23] that implements the ARM big.LITTLE architecture (4 big cores + 4 LITTLE cores), but in this work we only use the big cores. Big cores are equipped with a shared 2MB L2 cache. We profile the CPU using ARM Development Studio toolkit.

V. Results

Due to the efficiency of FC grouped convolution used in FC SqueezeBERT, the model better manages memory, specifically for big batch sizes, and results in up to 42% reduction in its fine-tuning time on the Jetson TX2. In the following, we will look at how main memory usage, cache utilization, and GPU initialization all contribute to this. Afterward, we present an ablation study and study the effect of our utilized FC grouped convolution in SqueezeBERT implementation in terms of inference latency. In the end, we study the efficiency of utilizing FC grouped convolution in other DNN models.

A. Reduced Main Memory Usage

PyTorch and CUDA libraries do not dedicate more than the needed memory in FC layers implementations [24]. This optimized memory allocation results in a considerable main memory (RAM) savings. However, this is not the case for 1D grouped convolution implementations. As illustrated in Fig. 3b (bars and the left axis), for example, we reduce RAM usage by 20% for a batch size of 32 on the Jetson TX2. Fine-tuning speeds up considerably as a result, as little to no memory swap operations are needed. Also, we observe that even though FC SqueezeBERT has the same number of trainable parameters as SqueezeBERT, it reduces the fine-tuning memory usage by 22.1%. This is the case since other than trainable parameters in the forward pass of training, there exist many other tasks that use memory during training, and they will not be optimized using FC SqueezeBERT (e.g., memory used for feeding inputs from a dataset to the model).

B. Cache Utilization Optimization

When we utilize FC layers rather than 1D grouped convolutions, low-level libraries do not try to keep data (e.g., input, filter weights, and activations) in high-speed memories, such as GPU L2 cache. This is the case since there is no data locality in the FC layers. As a result, the libraries can manage cache memory better. This better memory management results in fewer cache misses. As illustrated in Fig. 3b (green dashed line and the right axis), the cache miss rate of FC SqueezeBERT drops significantly (e.g., more than 15% for the batch size equal to 32). This drop is more significant for big batch sizes, where FC SqueezeBERT saves more memory by implementing 1D grouped convolutions using FC layers. Note that this saving translates to having less unused data in cache, and consequently, a smaller cache miss rate.

C. GPU Initialization Time Effect

The relative fine-tuning time reduction of FC SqueezeBERT is not constant across all downstream tasks. We measured the fine-tuning time of SqueezeBERT, FC SqueezeBERT, and BERT-base over downstream tasks and normalized with respect to the time required to fine-tune BERT-base As shown in Fig. 3c, FC SqueezeBERT's efficient cache and main memory management results in faster fine-tuning across all downstream tasks. However, some tasks benefit more than others. When fine-tuning time is on the order of minutes (e.g., MNLI and WNLI), GPU initialization time becomes significant. Note that GPU initialization time for these tasks is on the order of tens of seconds, and FC SqueezeBERT cannot optimize it. Therefore, we receive less relative fine-tuning time reduction ($\sim$27%) for those tasks. Moreover, we observe that for the largest tasks, such as RTE and SST2, we can reduce the fine-tuning time of SqueezeBERT about 42%. It is worth mentioning that, for all tasks, FC SqueezeBERT outperforms BERT-base.

D. Ablation Study

1) Cache Miss Rate and Memory Utilization: For each task, we allocated an empty memory block, in RAM, with the size of memory that FC SqueezeBERT saves from SqueezeBERT. Afterward, we run fine-tuning with batches of 32 over GLUE tasks. Via this experiment, we try to measure the contribution of each of our optimizations (memory and cache) to the fine-tuning time reduction. We demonstrate the portions of these contributions on the green bars of Fig. 3b. As is shown, FC SqueezeBERT cache access optimization constitutes between 12.8% to 17.4% of the entire fine-tuning time across different tasks. In other words, most of the fine-tuning time reduction comes from the main memory optimization: reducing off-chip memory accesses. Also, note that the next generation of the Jetson board (Xavier [22]) does not provide additional on-chip memory compared to the Jetson TX2. Therefore, our method is extendable to the next generation of this embedded board.

2) Batch Size: Moreover, we vary the batch size in fine-tuning to observe its effect on memory usage, fine-tuning speed, and accuracy. We observe that reducing the batch size from 32 (which is the recommended batch size for SqueezeBERT [2]) to 16 causes more than 17% memory saving for SqueezeBERT, and consequently, speeds up its fine-tuning by 21% on the Jetson TX2 board. This is the case since GPU memory utilization of the Jetson TX2 would be reduced with batch size = 16, which results in no less off-chip memory accesses during fine-tuning. Therefore, even though the batch

size is smaller (it takes more iterations to fine-tune on the input dataset), fine-tuning remaines faster. However, this speed-up is received in exchange for a 1.6% accuracy drop on the MRPC task for example. We observe a more severe accuracy drop by decreasing the batch size, and over all downstream tasks. Note that it has been studied and reported that fine-tuning must be performed with big batch sizes for NLP models [12], and reducing the batch size would damage the accuracy of the models. This is the case since the batch size determines the speed and stability of the learning process [12].

E. FC SqueezeBERT Inference

FC SqueezeBERT has a negligible effect on inference speed while not affecting its accuracy. We observed that the inference speed of FC SqueezeBERT is, on average, 5.6% slower than SqueezeBERT when we run it on Jetson TX2 CPU (a 64-Bit Quad-Core ARM® Cortex-A57). We think this originates from the way that Tensorflow-Lite implements 1D grouped convolution, and FC layers are used in SqueezeBERT, and FC SqueezeBERT, respectively. Note that this slowdown is negligible compared to the speed-up that SquuezeBERT archives: it is 4.3× faster than BERT-base [2].

Also, we observed no accuracy loss for FC SqueezeBERT, and there is no need to re-train FC SqueezeBERT. This is the case since we made neither architectural change to, nor approximation of, the original SqueezeBERT model; instead, we re-implemented the 1D grouped convolution function, used in SqueezeBERT, using FC layers to improve its fine-tuning speed.

F. Hardware Comparison

We observe the same trend when we execute FC SqueezeBERT and SqueezeBERT on the ARM big cluster of Hikey 970 [23]. In other words, even-though fine-tuning of such huge BERT models takes days on an embedded CPU, the behaviour is the same: FC SqueezeBERT is 28% faster than SqueezeBERT. Also, when we utilize our FC-based implementation of 1D grouped convolution in SqueezeBERT, we observe a considerable L2 cache miss rate reduction, on average >11%, for CPU. However, this cache miss reduction is smaller than what we observed for the Jetson TX2 embedded GPU. We think this is due to the bigger L2 cache of the CPU that it can fit more of (unoptimized) 1D grouped convolutions from the original SqueezeBERT implementation.

G. Generalizability

1D grouped convolution is a popular method of optimization not only in NLP models, but also in computer vision networks such as MobileNet [7] , ShuffleNet [8], and EfficientNet [9]. In this regard, we re-implemented used 1D grouped convolutions in these models using our introduced method (FC-based 1D grouped convolution) and in Pytorch. We observe that FC 1D grouped convolution reduces the memory footprint of these models by 38.1%, 28.3%, and 16.9% for MobileNet, ShuffleNet, and EfficientNet, respectively. Note that the difference in memory saving across these models comes from the fact that the architectures of these models are different, and they utilized grouped convolution with different numbers of groups. Therefore, the amount of memory that can be saved using FC grouped convolution is different as well.

VI. Related Work

Batch Size: Due to the restrictions of edge devices, on-device learning is performed using micro-batches. Many studies have been done to find the smallest possible size for the micro-batches [25], [26] that does not affect the accuracy of a model. Note that small batch sizes translate to a smaller memory footprint, and therefore, can be deployed efficiently on edge devices. These works try to find a certain point where the micro-batch estimations become accurate enough and adding more samples no longer reduces the required number of training steps. Finding an optimal micro-batch size is orthogonal to our approach, and as we showed in Fig. 1, reducing the batch size alone will not solve the fine-tuning (training) problem. Moreover, reducing the batch size affects the accuracy of models [12]. Note that our method does not reduce the batch size to manage 1D grouped convolution memory size.

Inference Optimization: Almost all optimization methods used for inference have a direct effect on reducing training latency if they are implemented efficiently. Note that the forward pass of training is inference. Notable among such approaches are: weight sharing [27], [28]; reducing parameter count [29], [4], [2]; and, simplifying model architecture [6], [10]. These methods mostly focus on inference, and do not study the effects on fine-tuning (training). In our work, we specifically focus on the on-device fine-tuning problem of SqueezeBERT.

Memory Optimizations: As models grow, so do their memory usage, and substantial memory optimizations are needed to: 1) avoid running out of memory, and 2) reduce the off-chip memory accesses. There are some techniques that address the max memory usage of a model during its training using approximation [30], [31], [32], [33]. However, this affects training and/or inference accuracy, while our method does not affect accuracy at all.

Convolution Optimization: Common methods for convolution are direct convolution, Winograd convolution [34], and utilizing GEneral Matrix to matrix Multiplication (GEMM) convolution. Winograd convolution involves mapping data into Fourier space to allow multiplications to become additions. However, domain transformation's overhead for 1D convolution does not let Winograd convolution beat direct convolution. GEMM convolution is especially popular on GPUs, where the input data is expanded and reshaped using a method known as im2col to utilize probable data locality in convolution. However, these methods are not applicable for 1D convolution where the size of filter and input are the same and there is no data locality.

Also, some past works [20], [35] tried to optimize the memory usage and/or the speed of convolution using input and filter data reshaping for input and filter data to exploit a higher data locality and divide data between processing cores better.

However, these techniques are not applicable to 1D grouped convolution that does not have any data locality to exploit.

VII. Conclusion and Future Work

In this paper, we showed the importance of memory management, in different levels of memory hierarchy (cache and RAM), for grouped convolution based DNN models. Also, we introduced a simple, but very effective, method to reduce the memory footprint of 1D grouped convolutions in PyTorch. We re-implemented 1D grouped convolutions using fully-connected layers. As an example case, we studied, in detail, how the FC-based implementation of 1D grouped convolution could improve the speed of SqueezeBERT fine-tuning by 42%. This is achieved by managing cache and the main memory usage better. Also, we showed that FC SqueezeBERT addresses the SqueezeBERT's fine-tuning problem across different embedded platforms (CPU and GPU). Furthermore, we reported the optimization capability of FC grouped convolution in DNN models other than SqueezeBERT. We showed that, for example, FC grouped convolution can reduce the memory footprint of MobileNet [7] by 38%.

In the future, we will target other BERT architectures such as MobileBERT [4], which has half of the trainable parameters of SqueezeBERT ($\sim$25M), but its on-device fine-tuning time is just, at most, 17% faster than our work (FC SqueezeBERT) on different down-stream tasks [13]. We hypothesize that MobileBERT suffers from poor memory management in its bottleneck blocks (i.e., sparse convolutional layers).

VIII. Acknowledgment

This work was supported in part by Huawei Technologies Canada Inc. through the McGill Edge Intelligence Lab, and in part by Fonds de Recherche du Québec–Nature et Technologies (FRQNT) Postdoctoral Research Scholarship.

References

[1] J. Devlin *et al.*, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2019. [Online]. Available: http://arxiv.org/abs/1810.04805

[2] F. Iandola *et al.*, "SqueezeBERT: What can computer vision teach NLP about efficient neural networks?" 2020. [Online]. Available: http://arxiv.org/abs/2006.11316

[3] A. Gholami *et al.*, "SqueezeNext: Hardware-aware neural network design," 2018. [Online]. Available: http://arxiv.org/abs/1803.10615

[4] Z. Sun *et al.*, "MobileBERT: a compact task-agnostic BERT for resource-limited devices," 2020. [Online]. Available: http://arxiv.org/abs/2004.02984

[5] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation." [Online]. Available: http://arxiv.org/abs/1603.01025

[6] M. Sandler *et al.*, "MobileNetV2: Inverted residuals and linear bottlenecks," 2019. [Online]. Available: http://arxiv.org/abs/1801.04381

[7] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," in *arXiv:1704.04861*, 2017.

[8] X. Zhang *et al.*, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *CVPR*, 2018.

[9] M. Tan *et al.*, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *ICML*, 2019.

[10] Z. Lan *et al.*, "ALBERT: A lite BERT for self-supervised learning of language representations." [Online]. Available: http://arxiv.org/abs/1909.11942

[11] A. e. a. Paszke, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[12] Does model size matter? a comparison of BERT and distilBERT. [Online]. Available: https://wandb.ai/jack-morris/david-vs-goliath/reports/Does-Model-Size-Matter-A-\Comparison-of-BERT-and-DistilBERT--VmlldzoxMDUxNzU

[13] A. Wang *et al.*, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," *arXiv:1804.07461 [cs]*, 2019.

[14] Understanding searches better than ever before. [Online]. Available: https://blog.google/products/search/search-language-understanding-bert/

[15] Z. Xiao and Y. Xiao, "Security and privacy in cloud computing," *IEEE Communications Surveys Tutorials*, vol. 15, no. 2, pp. 843–859, 2013.

[16] S. Wang *et al.*, "High-throughput CNN inference on embedded ARM big.LITTLE multicore processors," *IEEE TCAD*, vol. 39, no. 10, pp. 2254–2267, 2020.

[17] A. Ignatov *et al.*, "AI benchmark: All about deep learning on smartphones in 2019," in *2019 IEEE CVF Conference*.

[18] A. Vaswani *et al.*, "Attention is all you need," 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[19] P. Ganesh *et al.*, "Compressing large-scale transformer-based models: A case study on BERT," 2021. [Online]. Available: http://arxiv.org/abs/2002.11985

[20] M. Cho and D. Brand, "MEC: Memory-efficient convolution for deep neural network," in *PMLR*, ISSN: 2640-3498. [Online]. Available: https://proceedings.mlr.press/v70/cho17a.html

[21] "NVIDIA CUDNN, grouped convolution implementation details." [Online]. Available: https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html#grouped-convolutions

[22] X. Jetson. Deploy AI-powered autonomous machines at scale. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/

[23] "Hikey970," 2018. [Online]. Available: https://www.96boards.org/product/hikey970/

[24] C. F. Implementation. CUDA fully connected layer implementation. Archive Location: Optimizing Performance. [Online]. Available: http://docs.nvidia.com/deeplearning/frameworks/dl-performance-fully-connected/index.html

[25] C. J. Shallue *et al.*, "Measuring the effects of data parallelism on neural network training," 2019. [Online]. Available: http://arxiv.org/abs/1811.03600

[26] S. McCandlish *et al.*, "An empirical model of large-batch training," 2018. [Online]. Available: http://arxiv.org/abs/1812.06162

[27] S. Kim *et al.*, "I-BERT: Integer-only BERT quantization," 2021. [Online]. Available: http://arxiv.org/abs/2101.01321

[28] M. A. Gordon *et al.*, "Compressing BERT: Studying the effects of weight pruning on transfer learning," 2020. [Online]. Available: http://arxiv.org/abs/2002.08307

[29] X. Jiao *et al.*, "TinyBERT: Distilling BERT for natural language understanding," 2020. [Online]. Available: http://arxiv.org/abs/1909.10351

[30] M. Rhu *et al.*, "Compressing DMA engine: Leveraging activation sparsity for training deep neural networks," in *IEEE HPCA*, 2018.

[31] R. D. Evans *et al.*, "JPEG-ACT: Accelerating deep learning via transform-based lossy compression," in *IEEE ISCA*, 2020.

[32] A. N. Gomez *et al.*, "The reversible residual network: Backpropagation without storing activations," 2017. [Online]. Available: http://arxiv.org/abs/1707.04585

[33] S. H. Mozafari *et al.*, "Implementing convolutional neural networks using hartley stochastic computing with adaptive rate feature map compression," *IEEE OJCAS*, vol. 2, pp. 805–819, 2021.

[34] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *CVPR*.

[35] P. Gibson *et al.*, "Optimizing grouped convolutions on edge devices," in *IEEE ASAP*, 2020.