UNIVERSITY OF CALIFORNIA SAN DIEGO

Accelerating Attention Models on Hardware

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Electrical Engineering (Machine Learning and Data Science)

by

Zheng Li

Committee in charge:

Professor Mingu Kang, Chair
Professor Hadi Esmaeilzadeh
Professor Yatish Turakhia

2022

The thesis of Zheng Li is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2022

DEDICATION

To my family.

EPIGRAPH

Talk is cheap.
Show me the code.
—Linus Torvalds

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I would like to acknowledge my research advisor, Prof. Mingu Kang. I have never imagined achieving so much in my master study. My achievement today is not possible without your guidance in research, career and personal growth.

I am also grateful to my family and friends who supported me to finish two-year study during a global pandemic.

I would like to thank Professor Hadi Esmaeilzadeh, Dr. Amir Yazdanbakhsh, Soroush Ghodrati and Ashkan Moradifirouzabadi for their contribution in our research.

Chapter 2 is adapted from the material as it appears in Zheng Li, Soroush Ghodrati, Amir Yazdanbakhsh, Hadi Esmaeilzadeh, Mingu Kang. 2022. Accelerating Attention through Gradient-Based Learned Run- time Pruning. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. The thesis author was the primary investigator and author of this paper.

Chapter 3 is adapted from the material that has been submitted for publication as it appears in Amir Yazdanbakhsh, Ashkan Moradifirouzabadi, Zheng Li, Mingu Kang. "Sparse Attention Acceleration with Approximate In-Memory Pruning." The thesis author was the primary investigator and author of this paper.

VITA

2020	B. S. in Data Science, New York University Shanghai
2022	M. S. in Electrical Engineering (Machine Learning and Data Science), University of California San Diego

PUBLICATIONS

Zheng Li, Soroush Ghodrati, Amir Yazdanbakhsh, Hadi Esmaeilzadeh, Mingu Kang. 2022. Accelerating Attention through Gradient-Based Learned Run- time Pruning. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 14 pages.

ABSTRACT OF THE THESIS

Accelerating Attention Models on Hardware

by

Zheng Li

Master of Science in Electrical Engineering (Machine Learning and Data Science)

University of California San Diego, 2022

Professor Mingu Kang, Chair

The attention mechanism is the key to many state-of-the-art transformer-based models in Natural Language Processing and Computer Vision. The size of these models are growing rapidly while the computation and data movement cost and the on-chip memory demand is also growing beyond the capabilities of edge devices. This thesis provides solutions to address these challenges by developing strategies to prune the inconsequential attention scores efficiently and effectively. Attention score is the core of the attention mechanism in all transformer-based models. It measures the correlation of two tokens in a sequence. Low attention score value indicates unimportant correlation and minimal impact in subsequent calculation. Chapter 1 is the overall introduction of this thesis. In Chapter 2,

a novel gradient-based method to find the optimal threshold by to prune the inconsequential attention scores to reduce the computation cost is introduced. By training the model with the threshold, the optimal threshold value that maximizes the pruning rate and maintains model's accuracy is found. Based on the work in Chapter 2, an accelerator that features in-memory pruning of attention scores is introduced in Chapter 3 to reduce the data movement cost for attention models with long input sequence. Result shows these pruning strategies achieve high speedup, low energy consumption while maintain the accuracy across different transformer models on various benchmarks.

Chapter 1

Introduction

Natural Language Processing (NLP) is one of the most important domains in Artificial Intelligence and the key algorithm for many modern technological products, such as virtual assistants, real-time closed caption, etc. The fast advancement in these technology products was made possible by various transformer models. The core of these models is the self-attention algorithm. The attention algorithm was first introduced to the NLP domain to solve machine translation tasks [5]. The Transformer models was developed by making structural modifications to the previous model [58].

1.1 Attention in Transformer

The backbone of the attention mechanism across various transformer models is the same. A word in a sequence is first projected to a vector by an tokenizer and the resulting vector is called an embedding of the word. If we denote the length of the sequence as s and the size of the embedding as d_w, after tokenization, a sequence is projected to a matrix $\mathcal{X}$ of size $s \times d_w$. By multiplying this matrix with three different projection matrices of size $d_w \times d$, we get the key matrix $\mathcal{K}$, the query matrix $\mathcal{Q}$ and the value matrix $\mathcal{V}$ respectively as show in 1.1.

$$\mathcal{Q}_{s\times d} = \mathcal{X} \times \mathcal{W}^{\mathcal{Q}}; \quad \mathcal{K}_{s\times d} = \mathcal{X} \times \mathcal{W}^{\mathcal{K}}; \quad \mathcal{V}_{s\times d} = \mathcal{X} \times \mathcal{W}^{\mathcal{V}} \tag{1.1}$$

The attention score is defined as the product of the query matrix $\mathcal{Q}$ and the key matrix $\mathcal{K}$.

$$\text{Score}_{s\times s} = \mathcal{Q} \times \mathcal{K}^T \tag{1.2}$$

The attention score matrix shows the relevance between every two words in the sequence. For example, the element $Score_{ij}$ shows the relevance between word i and word j in the input sequence. Then, the $Score$ matrix is normalized by a factor of $1/\sqrt{d}$ to ensure the gradient is stable. [58] After normalization, each row of the attention score matrix is passed to a softmax function to transform the raw score values to probabilities.

$$\mathcal{P}_{s\times s} = \text{Softmax}(\text{Score}) \tag{1.3}$$

The final attention score is calculated by multiplying the softmax result and the value matrix $\mathcal{V}$.

$$\text{Att}_{s\times d} = \mathcal{P} \times \mathcal{V} \tag{1.4}$$

Different transformer models have different structures. But in general, the computation introduced above is a basic building block for all models and is called a head. Each attention layer has multiple heads with different weight and each attention model has multiple attention layers.

$$\text{Multi} - \text{Head Att}_{s\times d_w} = \text{Concat}(\text{Att}_1,\ \text{Att}_2,\ \cdots,\ \text{Att}_h) \times \mathcal{W}^o \tag{1.5}$$

Transformer models have become the state-of-the-art algorithm for natural language processing tasks since it was first proposed. The achievement of the transformer models is so extraordinary that not only the traditional deep learning algorithms in the NLP domain, such as RNN and LSTM, are abandoned by many researchers, but also in the Computer Vision (CV) domain, researchers' attention is shifting from the traditional Vision Convolutional Neural Networks (CNN) to the Transformer Models [27, 34, 38, 69, 26, 15].

1.2 Exploiting Run-time Pruning in Transformer models

The attention algorithm was developed to find the importance of all words in a sequence with respect to a single word. That is, how much attention is needed for every words when translating one specific word [5]. This way, the context is taken into account in translation. It is important to understand the context to accurately capture the meaning of a word in all NLP tasks. For example, in a financial news article, the meaning of the word 'Pound' is most likely to be currency unit. However, in other cases, the meaning of the word becomes a weight unit.

The attention algorithm understands the context of a sequence by calculating the attention score of every word in the sequence with respect to a word. The higher the attention score is, the more important a word is for understanding another word. It it intuitive that in a long sequence of text, only a few words are important to understand a specific word, while other words are nearly irrelevant. In Transformer models, smaller attention score values have little to none impact for the computation in the subsequent calculation steps. That is too say, we can expect the same result even if we ignore the unimportant words during computation. However, if we ignore too many words, our understanding might be distorted. So finding a good threshold to determine if a word is important or not becomes critical. In modern Transformer models, multiple attention scores in different attention layers to provide different perspectives in understanding a sequence. Thus, it is ideal to have a different threshold for every attention layer, so that no information is lost in all layers. At the same time, it is ideal that more smaller attention score values are pruned and more computation can be skipped.

In some recent studies, attempts have been made to find such a threshold to skip inconsequential computations and improve the efficiency of the models. But in these studies, the threshold are all empirical values found manually [20, 62, 21, 56]. And all the attention layers and attention heads have the same threshold value. Setting good values are highly dependent on the researcher's experience and effort

and there is no guarantee that the optimal threshold value can be found on different models and tasks, especially when all attention heads and attention layers have the same threshold value.

1.3 Gradient-based optimization

The method presented in this thesis, addresses the drawbacks of the empirical methods. The threshold is integrated in the loss function of the Transformer models as a regularization term. The detailed implementation will be explained in Chapter 2, this section is a brief introduction of the whole algorithm.

Modern machine learning and deep learning model training can be framed as a non-convex optimization problem. The loss function is defined to represent the target of the optimization. And the loss function is usually a multi-dimensional non-convex function with very complex shape. Given the complexity, it is almost impossible to calculate the theoretical optimal value. Thus, almost all the machine learning algorithm uses gradient-descent to find a point of minimum loss [47]. In transformer models, the cross entropy loss [42] and the Kullback-Leibler divergence [33] are two commonly used loss functions.

In machine learning, regularization terms are usually added to the loss function to prevent overfitting [32, 74, 52] and improve sparsity [17, 51, 3]. However, in order to use gradient descent to train the machine learning model, the regularization term added to the loss function has to be differentiable. In general, pruning is not a differentiable operation. So no attempt has been made before to use back-propagation to find an optimal value of the pruning threshold. In this work, the pruning operation is replaced by a differentiable soft-pruning operation, enabling gradient to flow across layers. The transformer models' parameters are optimized together with the pruning threshold by back-propagation to find the optimal threshold value for each attention layer in the model. Preserving the model accuracy and increasing the threshold value are two opposite objectives. However, with our method, by leveraging the loss function, the relative importance of these two objectives can be determined and an theoretically optimal threshold value

cam be found in all circumstances. Note that back-propagation is the fundamental method used for all machine learning models. So the method in this paper does not rely on prior experience like the previous empirical methods.

In this design, at runtime, the attention score values below the learned threshold are pruned by changing the value to $-\infty$. This way, the following computation can be skipped for the pruned values. A bit-serial architecture, called LeOPArd[1], is also proposed in this work to maximize the benefits by terminating the computation even before pruning the subsequent calculation. In stead of just pruning, the computation cost is reduced to the lowest bit level by this early termination design. The early termination algorithm is well designed such that the accuracy is guaranteed to be preserved even if the computation is finished much earlier. More specifically, it can be determined at bit level if the partial result of the dot-product is possible to exceed the threshold after the computation is completely finished.

The performance improvement from the computational and micro architectural design of LeOPArd is evaluated on different state-of-the-are transformer models using popular benchmarks in NLP and CV. On average, LeOPArd achieves 1.9× speedup and 3.9× energy reduction, compared to a baseline design without pruning and early termination support in an iso-area setting. To further study the improvement of LeOPArd, the contribution of runtime pruning and the bit-serial early terminating are evaluated separately. On average, for the 3.9× overall energy reduction of LeOPArd, 2.1× is brought by the runtime computation pruning algorithm and 1.8× comes from bit-level early termination. The result of LeOPArd evaluation is also compared to similar previous accelerators, A^3 [20] and SpAtten [62], which also implemented runtime pruning. The comparison results from LeOPArd shows the gradient-based soft pruning design finds the optimal threshold that brings great benefit in runtime pruning and early termination.

[1] **LeOPArd**: **L**earning thr**E**sholds for **O**n-the-fly **P**runing **A**cceleration of t**R**ansformer mo**D**els.

1.4 Data-movement optimization

Despite saving self-attention computation cost, the pruning strategy itself does not effectively address the main cost driver of the self-attention mechanism: data communication overhead. This is because identifying the relevance of key embeddings per query, especially to preserve model accuracy, still requires fetching all embeddings to on-chip resources and performing costly query−key computations. Commonly, these methods presume sufficiently large on-chip resources to keep all embeddings on chip. As the input sequence of the model keeps growing, this assumption does not hold for some mobile and edge devices. For example, if we embrace a design with only 20% of requisite on-chip buffers available for embeddings, data communication emerges as the main determinant of efficiency (on average, $>60\%$ of total energy consumption as shown in Figure 3.1). To address this, an in-memory pruning stragety that obviate the need to bring embeddings onto the chip is described in Chapter 3.

An emerging body of work has illustrated significant benefits of ReRAM in-memory computing, due to the inherent efficiency of analog computing and massive parallelism capability [19, 67, 71, 63, 50, 12, 36, 41, 49]. We leverage ReRAM technology to enable in-memory pruning, reducing the pressure on the accelerator to fetch all embeddings onto the chip. While appealing, materializing the possibility of in-memory pruning comes with its own challenges, listed as follows:

- Circuit inaccuracies: There are various inaccuracies, such as thermal and coupling noise, associated with ReRAM analog circuitry, which limit the precision of in-memory computing.

- Data conversion overhead: Runtime pruning [37], a common approach to preserve model accuracy, requires layer-wise comparisons with a threshold value. The cost of converting the analog results of in-memory computing (multiple bits) to the digital domain for perpetual comparisons against threshold values can outweigh the benefits of in-memory computing.

- Selective read of unpruned embeddings: Supporting in-memory ReRAM pruning enforces a particular data layout for key embeddings. However,

this layout constraints the ability to selectively read the unpruned vectors.

The detailed implementation will be explained in Chapter 3. As a brief introduction, the concerns mentioned above are solved in the following ways:

1. For circuit inaccuracies, we introduce a unique perspective on the ReRAM in-memory computing paradigm. We employ approximate in-memory compute and precise on-chip recompute in together to avoid performance degradation caused by this issue.

2. For data conversion overhead, we employ analog comparators to carry out the comparisons with threshold values and instead produce 1-bit data to indicate the pruning status. With this shift in design, we reduce the hardware cost, which is proportional to input bit precision, to merely the cost of a series of 1-bit analog to digital converters (ADCs).

3. For selective read of unpruned embeddings, we repurpose an existing solution, which enables us to implement data reuse based on our observations. On the hardware side, we rely on recent taped-out transposable ReRAMs [60] that introduce in-situ transposed read access. While initially intended for efficiently accessing neural network weights, our application of this hardware selectively reads unpruned embeddings. For the data reuse, we observe that there is a considerable spatial locality between unpruned key vectors of adjacent queries. We exploit this spatial locality to improve data reuse and further reduce the data communication overhead.

We evaluate our approach SPRINT in several self-attention models with large sequences, including BERT, ALBERT,ViT, GPT-2 [27, 34, 15], and two futuristic designs (e.g. 2K and 4K input sequence length). Under an iso design, our results show that, on average, SPRINT delivers 7.5× speed-up and 19.6× energy reduction compared to a baseline design with 16KB on-chip memory. The benefit increases as on-chip resources become scarcer, representing a design point for resource constrained platforms, e.g. 1.6× more energy reduction with 16KB on-chip memory than the case with 64KB capacity. Under an iso design, our results show that,

on average, SPRINT delivers 7.5× speed-up and 19.6× energy reduction compared to a baseline design with 16KB on-chip memory. The benefit increases as on-chip resources become scarcer, representing a design point for resource constrained platforms, e.g. 1.6× more energy reduction with 16KB on-chip memory than the case with 64KB capacity.

Chapter 2

Attention Mechanism in Natural Language Processing

Most of the computation in transformer models are from their most basic building block, attention layers. From the calculation steps introduced above, it is obvious that the majority of the computation cost comes from the inner product of the Query matrix and the Key matrix (Equation 3.4) and the inner product of the Score matrix and the Value matrix (Equation 1.4), both of which are the multiplication of two matrices with $s \times d$ dimensions. And the time complexity of both multiplications is $\mathcal{O}(s^2 d)$. If these computation steps can be save in one attention layer, given the large number of layers in modern transformer models, a significant portion of the total computation can be saved just by runtime pruning.

2.1 Algorithmic Design

The section introduces the algorithmic design of LeOPArd for soft threshold learning, runtime pruning and early termination. As has been introduced in Chapter 1, the challenge of finding the optimal threshold value is the differentiablity issue of the pruning operation, which is solved by our first algorithmic design. The first algorithmic design introduced is the differentiable soft threshold pruning technique. This method eliminates inconsequential attention layer computations as early as possible, right after the multiplication of $\mathcal{Q}$ matrix and $\mathcal{K}$ matrix, to ac-

celerate the computation. Particularly, this algorithm sets the pruning thresholds, which is a different value for each attention layer, as parameters to be learnt together with other model parameters in retraining. Both the task specific fine-tuned model parameters and the pruning threshold are adjusted in the retraining process to increase the model sparsity while maintaining the accuracy. After the layer-wise thresholds are learned, the $Score = \mathcal{Q} \times \mathcal{K}^T$ values are compared against the learned pruning thresholds per attention layer and prunes the ones that are lower than the learned threshold. Different from the learned weight pruning method for image classification models [3], the pruning threshold values in our work depends on the specific transformer model and the particular task. As a result, the sparsity achieved by this algorithm varies from task to task and model to model. Because of the content-specific nature of our pruning method, very high sparsity is achieved in the attention layer computations while yielding almost no accuracy loss.

2.2 Learned Per-Layer Pruning

There are three major challenges in designing a algorithm to learn the optimal pruning threshold for each layer. The first challenge is, each pruning threshold can take any value from $-\infty$ to ∞. However, it is common for a modern transformer model to have a large number of layers. For example, the Bert Large model, BERT-L has 24 layers. Then, there are 24 values, each can take any value, to be learned. Second, as previous studies has found [20, 62], simply searching for the threshold values only could be detrimental to the models' accuracy, which is the reason that previous studies used empirical values as threshold. Training the model parameters together with the threshold could potentially solve the challenges mentioned above. However, the pruning operation is not a differentiable. So we cannot use train it directly using gradient descent. Also, if the loss function of the model is not updated, the threshold value is bound to become infinity because that way the model accuracy is the highest. To solve these new problems, the original pruning operation is replaced by a new soft threshold pruning function. And a surrogate regularizer is added the loss function to increase the sparsity of the

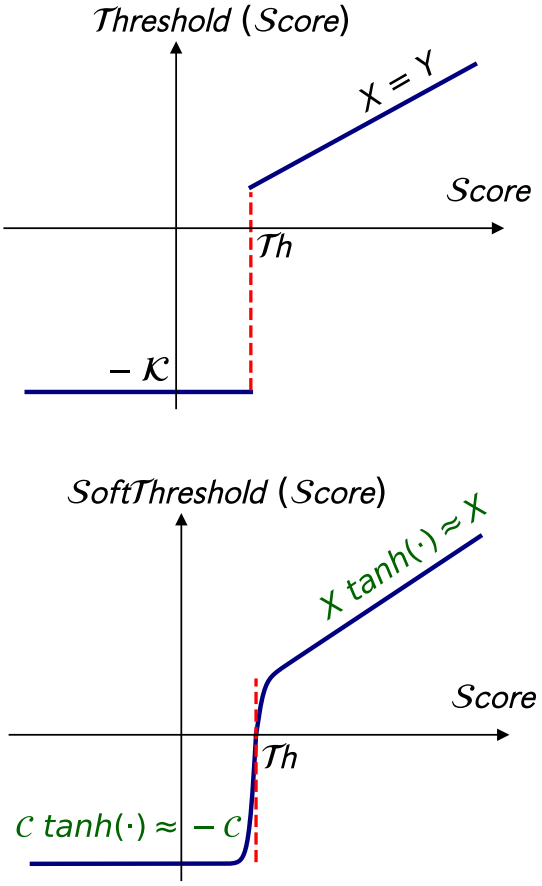


Figure 2.1: Pruning operation on attention *Score*: (a) ideal magnitude-based pruning, (b) proposed differentiable pruning operation with soft threshold.

model. In the following paragraphs, the two algorithmic design principles, namely "*pruning with soft threshold*" and "*surrogate L_0 regularization*" is introduced.

2.2.1 Pruning with soft threshold.

The original pruning operation for *Score* values (e.g. $Score = \mathcal{Q} \times \mathcal{K}^T$ is shown in Figure 2.1 (a), where $\mathcal{Q}$ and $\mathcal{K}$ are d-dimension vectors corresponding to a single word in the input sequence). The *Score* values greater than the threshold, $\mathcal{T}h$, remain unchanged and those less than $\mathcal{T}h$ are changed to a negative number with large a absolute value. The "*softmax(·)*" operation maps a set of values to probability values, larger positive values are mapped to probability values closer to 1 and negative values are mapped to probability values closer to zero. Changing

the value to a large negative number makes the corresponding values become zero after the "*softmax(·)*" operation. This way, these values under the threshold are pruned out and the subsequent calculation is not needed anymore. But as shown in Figure 2.1 (a), this pruning operation is not continuous and thus not differentiable. Current gradient descent optimization cannot be applied directly in this setting.

To bypass the difficulty caused by the original pruning operation, an approximate function is used to replace the original pruning operation. This approximate function uses a soft threshold (shown in Figure 2.1) as follows:

$$\text{SoftThreshold(x)} = \begin{cases} x \tanh(s(x - \mathcal{T}h)), & x \geq \mathcal{T}h \\ c \tanh(s(x - \mathcal{T}h)), & x < \mathcal{T}h \end{cases} \tag{2.1}$$

In this equation, the value of s controls the sharpness of the curve of this function. If the value of s is larger, the curve becomes sharper and closer to the original pruning operation's curve. However, it's still continuous. Thus it is differentiable and gradient descent does work in this case. Supporting the learning gradients to flow at the vicinity of $\mathcal{T}h$ allow the gradient-based learning algorithm to either push down the model parameters (e.g. $\mathcal{Q}$ and $\mathcal{K}$) below the threshold or lift them above the threshold according to their contributions to the overall model accuracy.

From Figure 2.1, we can see that if the input value is larger than the threshold $\mathcal{T}h$, the output of the soft threshold function is asymptotically close to the input value itself. And if the input is less than the threshold, the output is just $-c$. Except a small range near the threshold, the soft threshold function approximates the original pruning function. In most cases, the difference in the output of the two functions are negligible. In our experiments, we empirically find that setting $c = 1000$ and $s = 10$ yield a good approximation for pruning and enables robust training.

2.2.2 Differentiable surrogate L_0 regularization.

Applying the soft threshold function solves the problem of gradient. However, the concern of sparsity is not still not addressed. Obviously, the original model's

loss function only measures the accuracy of the model. So the only goal of the optimization steps is to improve the accuracy of the model. Then it is possible that the value of the threshold continuously drops in the training process so that the model's accuracy can be maximized. However, lowering the threshold value may cause the sparsity after the pruning to drop as well. Imaging if the threshold value is too low, that no value in the score matrix is pruned out and the sparsity is zero, the pruning operation just becomes meaningless. To tackle this, a regularization term is added to the loss function. This is a commonly used method to prevent model over-fitting in machine learning. To increase the sparsity of the model, a L_0 regularizer is used on the model parameters. The updated loss function is shown in the equation below:

$$L_{tot}(\theta) = \frac{1}{N}\Big(\sum_{i=1}^{N}\mathcal{L}\big(A(x_i;\ \theta),\ y_i\big)\Big) + \lambda||\theta||_0 \tag{2.2}$$

$$\lambda||\theta||_0 = \sum_{j=1}^{|\theta|}1[\theta_j \neq 0] \tag{2.3}$$

Here, $\mathcal{L}$ is the original loss of the model before the update, where $A(\cdot)$ is the model's prediction for the input x_i when the parameters are θ and y_i is the true value. This part of the loss function measures the accuracy of the model. $||\theta||_0$ is the L_0 regularization term. 1 in 2.3 is a function that counts the number of non zero elements in the model parameters. λ is a weight that controls the importance of this regularization term.

However, like the original pruning operation, this L_0 regularization term is not differentiable as well. Louizos et al. [40] sloved a similar problem by reparameterization of model parameters in order to take the gradients. This method worked well on Wide Residual Networks [72] and small datasets, but it has been shown that it fails to work as well on larger models and more intensive tasks. To solve this problem in this particular case, an alternative representation of the regularization term is used to bypass the non-differentiable issue. Here, instead of simply counting the number of values after pruning, a Sigmoid function, as shown in equa-

tion 2.5, is applied to approximate the counting function and enable the gradient calculation, similar to the soft pruning method introduced above.

$$||\theta||_0 = \sum_{j=1}^{|score|} 1[\text{score}_j > -c] \tag{2.4}$$

$$||\theta||_0 \approx \sum_{j=1}^{|score|} \text{sigmoid}(k(\text{score}_j + c - \alpha)) \tag{2.5}$$

In all experiments, k is set to 100 and α is set to 1. This way, the sigmoid$(\cdot)$ function is very close to one for input values larger than $-c$ and close to zero when the input value is smller than $-c$. Thus, the output of equation 2.5 is almost the same as the output of equation 2.4.

2.2.3 Pruning mechanism

After solving the problem of pruning with differentiable operation to enable threshold learning and adding the L_0 regularization term to increase the model sparsity, the fine-tuned models are retrained with the two techniques applied. Note that the retraining step is task-spesific and model specific. For example, a model like the Bert base, BERT-B, is first fine-tuned on a specific task dataset, like SQUAD. Then the soft threshold pruning and regularization term is added to the model and the model is trained again on the same dataset. In the training process, both the threshold value and other model parameters are slightly adjusted based on the gradient descent rules enabled by the soft threshold pruning. As a result, in the Score matrix, the important values become larger relative to the threshold $\mathcal{T}h$ after training, so that they are not pruned out. On the other hand, unimportant values in the Score matrix are made smaller relative to the threshold $\mathcal{T}h$ so that they are pruned and the model sparsity is increased. Using this method, the model parameters are updated according to the gradient to change which values in the Score matrix are kept after pruning and which are pruned after pruning. The threshold value $\mathcal{T}h$ were initialized as zero at the beginning of the retraining

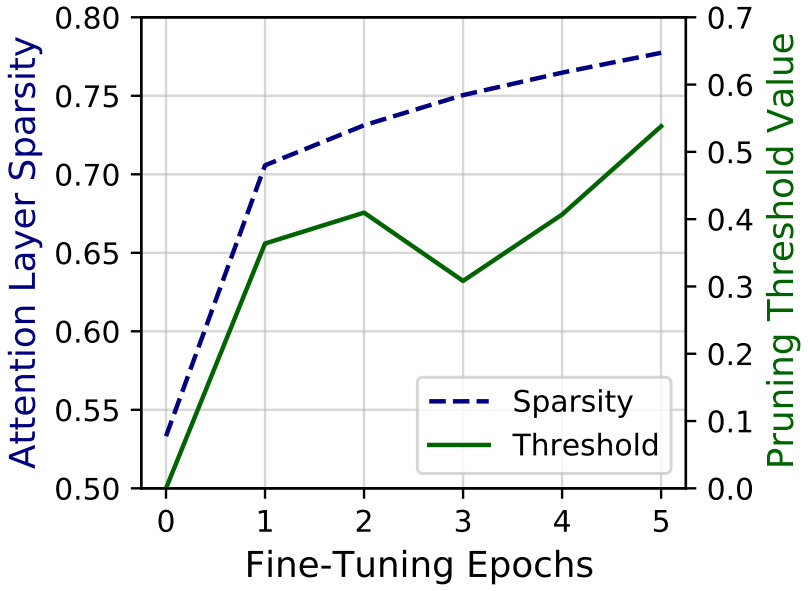


Figure 2.2: An example of attention layer sparsity and its corresponding pruning threshold values for BERT-L model on QNLI task from GLUE benchmark.

process. And the retraining process is five epochs for all the models and tasks evaluated in this study.

The retraining is proven to be highly effective. As shown in Figure 2.2, the threshold value increases over the process of training and correspondingly the model sparsity increases as well, which is the expected outcome after adding the L_0 regularization term. In epoch 3, despite the threshold value slightly drops, the model sparsity still increases. This suggests the benefit of training both the threshold and other parameters together. Even though the threshold declines, other parameters are adjusted to increase the model sparsity. It can also be observed from Figure 2.3 that during the five epochs of training process, the overall loss is steadily decreasing and gradually converging to an optimal spot. Both of the figures shows the training process of retraining BERT-L model on the QNLI task from the GLUE benchmark.

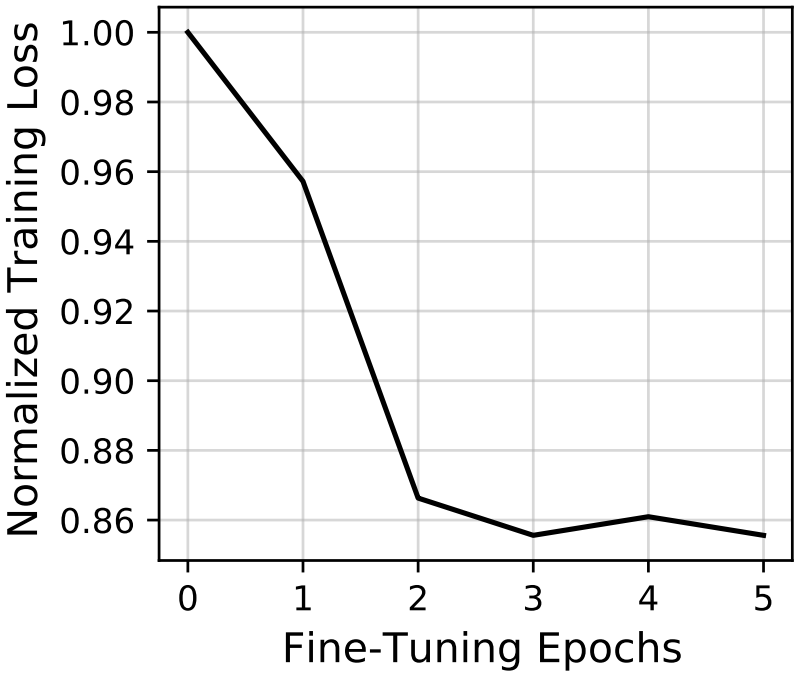


Figure 2.3: An example of normalized training loss as fine-tuning epochs progress for BERT-L model on QNLI task from GLUE benchmark.

2.2.4 Bit-Level Early-Compute Termination

The learned pruning offers a unique opportunity to further improve the LeOPArd performance through bit-serial $\mathcal{Q} \times \mathcal{K}^T$ computation. After getting the optimal pruning threshold values through the retraining process, it is ideal to further exploit the high model sparsity achieved, so that the performance of LeOPArd can be further improved. If the LeOPArd is aware that the value of the $\mathcal{Q} \times \mathcal{K}^T$ product matrix will never reach the pruning threshold found in retraining, the bit-serial computation can be stopped earlier to further reduce the cost. But stopping the computation early is not trivial. Because if applied incorrectly, it may be detrimental to the model's accuracy because of the false termination, which is against the purpose of developing the LeOPArd system.

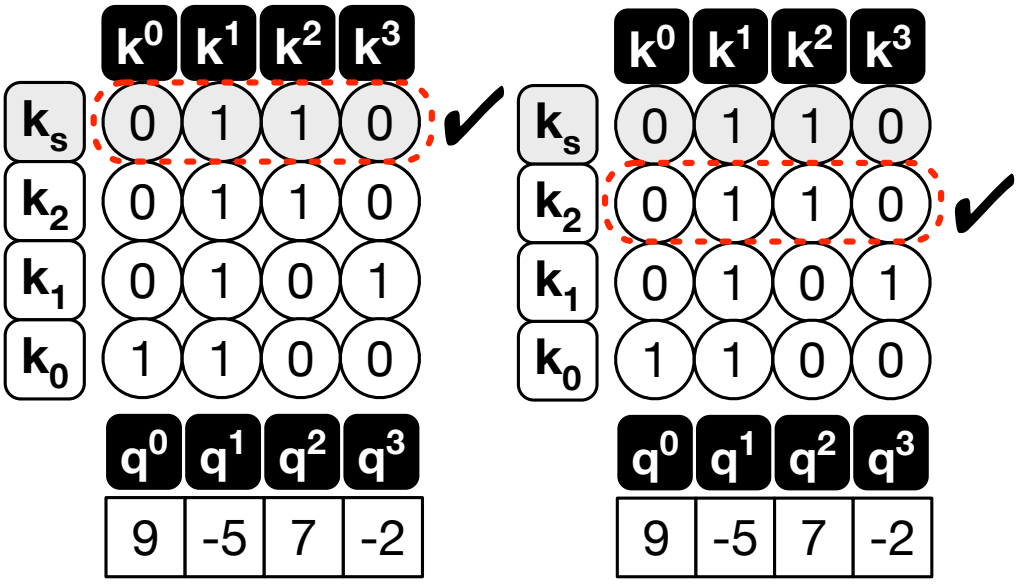
In order to make sure the termination does not change the final result of the computation, a dynamically adjusted conservative margin value is calculated to compensate for the part that has not been calculated in the bit-serial computation

step. The purpose of adding this conservative margin is to judge whether the value will exceed the threshold given the maximum value for the unseen bits. If the LeOPArd system finds that even after adding the margin to the current result, the final result will still be lower than the learned threshold, the computation is terminated at this step. The value in the $\mathcal{Q} \times \mathcal{K}^T$ product matrix is pruned and the computation for the remaining bits are skipped. In the following paragraph, the calculation of early-compute termination with a conservative margin will be explained.

Early-compute termination for dot-product operation.

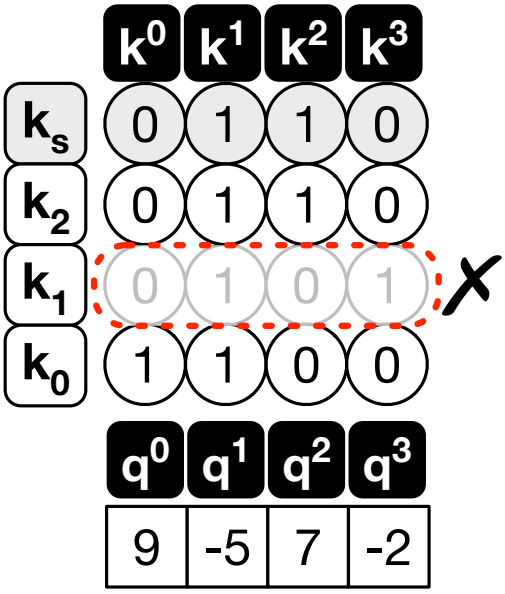
In Figure 2.4, an example of the computation process of the $\mathcal{Q} \times \mathcal{K}^T$ inner product is shown. The bit-level representaion of $\mathcal{K}$ is displayed vertically from MSB $\rightarrow$ LSB. The $\mathcal{Q}$ values are stored in full-precision fixed-point format. For simpler illustration, this Figure assumes the computation is performed in sign-magnitude form, where k_s represents the sign-bit for $\mathcal{K}$ vector, and the absolute values of $\mathcal{K}$ elements are less than one.

In the first cycle, the elements with the same signs, for example, (k^0, q^0) and (k^1, q^1), are used to calculate the first margin. The reason behind this design is simple: only the multiplication of numbers with the same signs yields positive number and contribute to a larger product result in the end. However, two numbers with different sings are ignored because the margin because it only contributes negatively to the final result, which is against the conservative design of the margin. As shown in the table 2.1, both the product of k^2 and the q vector and the margin are updated. The margin is changed according to the largest possible positive contribution to the final value. In the second cycle, because the sum of $\mathcal{P}_2$ and $\mathcal{M}_2$ is below the threshold, the computation just ends here and the following cycles (highlighted in gray) are no longer performed. Note that, with this conservative margin design, it is guaranteed that the result of the pruning will be the same with the calculation without early termination.

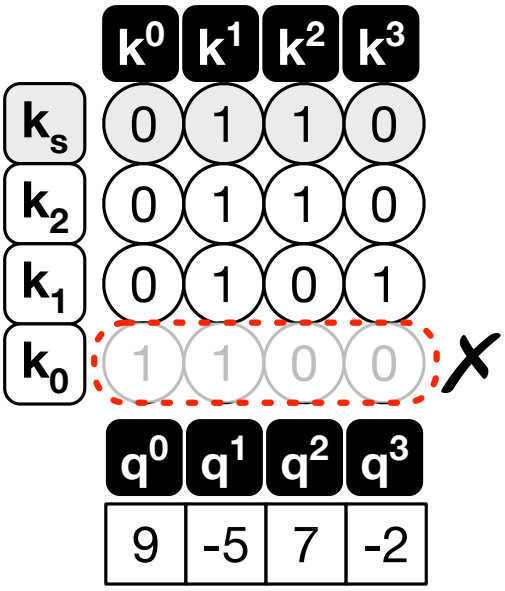


(a) Cycle = 1

(b) Cycle = 2



(c) Cycle = 3



(d) Cycle = 4

Figure 2.4: High-level overview of early-compute termination for dot-product operation. K_s indicates the sign bit. For simplicity, $\mathcal{K}$ elements are scaled to be between -1.0 and +1.0. The table shows the partial sum values after each cycle.

Table 2.1: the partial sum values after each cycle in Figure 2.4

Cycle	$\mathcal{P}$ **= Partial Sum**	$\mathcal{M}$ **= Conservative Margin**	**Early Termination?** ($\mathcal{T}h$ **= 5)**
1	$\mathcal{P}_1 = 0$	$\mathcal{M}_1 = (9+5)(2^{-1}+2^{-2}+2^{-3}) = 12.25$	$\mathcal{P}_1 + \mathcal{M}_1 = 12.25 \geq 5$;
2	$\mathcal{P}_2 = \mathcal{P}_1 + (5-7)2^{-1} = -1$	$\mathcal{M}_2 = (9+5)(2^{-2}+2^{-3}) = 5.25$	$\mathcal{P}_2 + \mathcal{M}_2 = 4.25 < 5$;
3	$\mathcal{P}_3 = \mathcal{P}_2 + (5-2)2^{-2} = -0.25$	$\mathcal{M}_3 = (9+5)(2^{-3}) = 1.75$	$\mathcal{P}_3 + \mathcal{M}_3 = 1.5 < 5$;
4	$\mathcal{P}_4 = \mathcal{P}_3 + (9+5)2^{-3} = 1.5$	$\mathcal{M}_4 = 0$	$\mathcal{P}_4 + \mathcal{M}_4 = 1.5 < 5$;

2.3 Hardware Architecture of LeOPArd

The hardware architecture of LeOPArd is designed in consideration of the following factors to exploit the algorithmic advancements introduced above:

- Localize the unpruned *Score* values and their corresponding positions in the output matrix utilizing the learned threshold value.
- Reduce the computation and memory access by implementing the bit-serial processing strategy to terminate the computation early.
- Skip the pruned $Score \times \mathcal{V}$ operation and maintain high compute utilization at the same time

2.3.1 Overall Architecture

Since the available parallelism in multi-head attention layers, a tile-based hardware architecture is designed for LeOPArd, where attention heads are partitioned across the tiles, and the operations in the tiles are independent of each other on their corresponding heads. The high-level microarchitecture of a single LeOPArd tile is show in Figure 2.5. There are two modules in each tile to conduct the computation in the attention layer:

- A front-end unit, named Query Key Processing Unit (QK-PU). This unit reads $\mathcal{Q}$ row vectors from the Q matrix from the off-chip memory in a streaming fashion and it reads the $\mathcal{K}$ matrix from a local buffer, and conducts vector-matrix multiplication between a $\mathcal{Q}$ vector and a $\mathcal{K}$ matrix. Here, every $\mathcal{Q}$ row vectors is corresponds to a single word/token from the raw input to the model. There is also a 1-D array of bit-serial dot-product unit, QK-DPUs,,

in this unit. The QK-DPU implements the early termination algorithm according to the pruning threshold and locates the unpruned values in order to passing it to the next stage.

- A back-end unit, named Value Processing Unit (V-PU). This unit conducts the softmax operation on the *Score* matrix after pruning followed by the $Score \times \mathcal{V}$ multiplication to generate the final result.

A group of FIFOs that stores the unpruned *Score* values and their positions connect the front-end unit and the back-end unit. Multiple (N_{QK}) QK-DPUs in front-end shares a single V-PU in the back-end, since the pruning rate is high enough, even one V-PU can handle multiple QK-DPUs in front-end. In the case of the $\mathcal{Q} \times \mathcal{K}$ is finished in the front-end, but the back-end is still processing the last $\mathcal{Q}$ vector, then the frond-end unit has to pause until the back-end finishes the computation.

In this design, the choice of the N_{QK} is crucial for maximizing the overall throughput and resource utilization in the back-end. In this study, both $N_{\mathrm{QK}} =$ 6 and 8 are selected to in order to get area efficiency and higher utilization, respectively. All the $\mathcal{K}$ and $\mathcal{V}$ matrices are read from off-chip memory and stored on on-chip buffers before any multiplication starts, while the $\mathcal{Q}$ vectors are steamed in. This way, the DRAM costs are amortized because the vectors can be re-used by the number of sequence elements (e.g. 512 in BERT).

2.3.2 Online Pruning Hardware Realization via Bit-serial Execution

To exploit the benefit from the algorithm innovation in threshold learning and sparsity improvement, the frond-end design shown in Figure 2.5-(a) as a collection of bit-serial dot-product units (QK-DPU).

Overall front-end execution flow. For the $\mathcal{Q} \times \mathcal{K}$ multiplication in the front-end unit, the $\mathcal{Q}$ vectors are streamed in from Q-FIFO and then distributed to each QK-DPU To perform the *Score* computations, the $\mathcal{Q}$ vectors are read sequentially from Q-FIFO and then broadcasted to each QK-DPU, while each QK-DPU reads

a $\mathcal{K}$ vector from its local Key Buffer and performs a vector dot-product operation. As such, while the $\mathcal{Q}$ vector is shared amongst the QK-DPUs, the $\mathcal{K}$ matrix is partitioned along its columns and is distributed across the Key Buffers. Each QK-DPU performs the dot-product operations in a *bit-serial* mode, where the $\mathcal{K}$ elements are processed in bit-sequential manner and the $\mathcal{Q}$ elements are processed as a whole (e.g. 12 bit). Whenever each QK-DPU finishes the processing of all its $\mathcal{K}$ bits for unpruned *Score*s or early terminates the computation due to not meeting the layer pruning threshold based on the margin calculation described in Section 2.2.4, it proceeds with the execution of next $\mathcal{K}$ vector. If a QK-DPU detects a unpruned *Score*, it stores the *Score* value and its corresponding index on Score-FIFO and IDX-FIFO, respectively, to be processed by the back-end unit later. Once all the QK-DPUs finish processing all their $\mathcal{K}$ vectors, the QK-PU reads the next $\mathcal{Q}$ vector from Q-FIFO and starts its processing.

Bit-serial dot-product execution. Figure 2.6-(a) depicts the microarchitectural details of our Bit-Serial Dot-product Engine (BS-DPE). The BS-DPE is a collection of Multiply-ACcumulate (MAC) units and it performs a 12-bit$\times\mathcal{B}$-bit dot-product operation per cycle, where the $\mathcal{Q}$ vector is kept in a local register and $\mathcal{K}$s are read from the Key Buffer $\mathcal{B}$-bit at a time in a sequential mode. We chose $\mathcal{B}$ = 2-bit as opposed to conventional bit-by-bit serial designs as the number of bits processed per cycle opens a unique trade-off space for the design of LeOPArd. Increasing the bits leads to better power efficiency due to less frequent latching of intermediate results, however it may degrade the performance as it reduces the resolution of bit-level early termination. As such we perform a design space exploration (Figure 2.15 in Section 2.4.4) and chose 2-bit serial execution as it strikes the right balance between power efficiency and performance. The BS-DPE accumulates all the intermediate results in around 20 bits to keep required precision of the computations. The output of the last 2-bit$\times$12-bit MAC unit then goes to a shifter to scale the partial results according to the current $\mathcal{K}$ bit position and is accumulated and stored in a register that holds the (partial) results of *Score* computations.

Pruning detection via dynamic margin calculation. As discussed in

Section 2.2.4 and Figure 2.4, to detect whether a current *Score* needs to be pruned and corresponding computations be terminated, QK-DPU dynamically calculates a *conservative upper-bound margin* ($\mathcal{M}$) and adds it with the current dot-product partial sum ($\mathcal{P}$) to compare it with the layer threshold ($\mathcal{T}h$). Figure 2.6-(b) and (c) show the details of hardware realization for margin calculation and thresholding logic, respectively. To calculate the margin according to Table in Figure 2.4, the margin calculation module first detects the $\mathcal{Q}$ and $\mathcal{K}$ pairs in the dot-product that yield positive product. To do so, during the processing of $\mathcal{K}$'s MSBs, the sign bits of $\mathcal{Q}$s and $\mathcal{K}$s are XORed. Only if the result is positive (XOR = 0), the absolute values of the corresponding $\mathcal{Q}$ are summed up to calculate the margin (e.g., resulting in $(9 + 5)$ in the Table of Figure 2.4). The summation result is stored in a Sum Register. Then, it is scaled by the fixed number, largest positive value (e.g. 0111...), which corresponds to $(2^{-1}+2^{-2}+2^{-3}+...)$ in Figure 2.4, storing $(9+5)(2^{-1}+2^{-2}+2^{-3}+...)$ in the margin register. On the other hand, if it turns out the multiplication yields a negative value (XOR = 1) during the processing of $\mathcal{K}$ MSBs, the computation is excluded and skipped toward the margin calculation, which is enabled via multiplication by zero (000...). The margin needs to be calculated dynamically for each bit position during bit-serial execution (such as $\mathcal{M}$ changing in each row of the Table in Figure 2.4). This is enabled by subtracting the shifted version of Sum Register value from the current margin in the margin register, e.g., $(9+5)(2^{-1}+2^{-2}+2^{-3}+...)-(9+5)(2^{-1}) = (9+5)(2^{-2}+2^{-3}+...)$ in the second row of the Table in Figure 2.4. This operation is iterated every bit position to generate the values in the subsequent rows of the Table in Figure 2.4. Note that, the margin calculation is a scalar computation (mostly shift and subtraction), which is amortized over the $d = 64$ dimension vector processing, incurring virtually no overhead.

Note that, since the Queries need to be multiplied by either of the three aforementioned bit streams, these multiplications are implemented merely with Shift and Add, instead of complex and expensive multipliers. Finally, the accumulated result of these operations yields the current margin.

After each cycle of the bit-serial operation, the thresholding module (Figure 2.6-

(c)) adds the updated partial sum with the current margin and compares it with the layer threshold $\mathcal{T}h$ to determine the continuation of the dot-product or its termination for pruning of the current *Score*.

Final score index calculation. The QK-DPU calculates the indices of the unpruned *Score*s using a set of two counters, as shown in Figure 2.6-(d). First, Bit-serial Cntr increments with the number of bits processed by the QK-DPU and gets reset whenever it reaches its maximum (i.e. 6 (= 12bit/$\mathcal{B}$)) for processing all bits for unpruned *Score*s) or the Early stop flag is asserted. Second, the value of IDX Cntr shows the position of the current *Score* in the vector and increments whenever the Bit-serial Cntr gets reset, ending the computation of that *Score*. Finally, if the IDX Cntr increments and the Early stop flag is low, the QK-DPU pushes the content of this counter to IDX FIFO, because it means that the corresponding *Score* is not pruned and will be used for further processing in the V-PU.

2.3.3 Back-End Value Processing

As shown in Figure 2.5-(b), the LeOPArd tile's back-end stage, V-PU, consumes the unpruned *Score*s and executes the Softmax operation, followed by multiplication with $\mathcal{V}$ vectors and finally storing the results to an Output-FIFO. Whenever the Score-FIFO is not empty, the V-PU starts the Softmax operation (e^x and accumulation) to calculate the probabilities. We implemented the Softmax module of V-PU similarly to the Look-Up-Table (LUT)-based methodology in A^3 [20]. Whenever the output probability is produced, the V-PU uses the indices of the unpruned *Score*s to read the corresponding $\mathcal{V}$ vector. Finally, the $\mathcal{V}$ vector is weighted by the output of the Softmax module with a 1-D array of MAC units. The elements of $\mathcal{V}$ vector are distributed and the probabilities are shared across the MAC units, similar to a 1-D systolic array. With such design, the V-PU consumes the *Score*s sequentially to complete the weighted-sum of $\mathcal{V}$ vectors, and accumulates the partial results over multiple cycles while only accessing the unpruned $\mathcal{V}$ vectors. As such, it rightfully leverages the provided pruning by the front-end stage and eliminates the inconsequential computations.

Table 2.2: Microarchitectural configurations of a LeOPArd tile.

Hardware modules	**Configurations**
QK-PU	6 / 8 QK-DPU (=N_{QK}), each 64 (=$\mathcal{D}$) tap 12×2 bit-serial
Key Buffer	48KB in total (= 8KB×6 / 6KB×8 banks), 128-bit port per bank
V-PU	Single 1-D 64 (=$\mathcal{D}$) way 16×16-bit MAC array
Value Buffer	64KB (= 8KB × 8 banks), 128-bit port per bank
Softmax	24-bit input, 16-bit output, LUT: 1 KB
Score and IDX FIFOs	24-bit × 512 depth for Score, 8-bit × 512 depth for IDX

2.4 Evaluation

2.4.1 Methodology

Workloads. We evaluate LeOPArd on various NLP and Vision models: BERT-Base (BERT-B) [27], BERT-Large (BERT-L) [27], MemN2N [55], ALBERT-XX-Large(ALBERT-XX-L) [34], GPT-2-Large (GPT-2-L) [45], and ViT-Base(ViT-B) [15]. To evaluate these models, we use five different datasets: (1) Facebook bAbI, which includes 20 different tasks [64] for MemN2N, (2) General Language Understanding Evaluation (GLUE) with nine different tasks [61] for BERT models, (3) Stanford Question Answering Dataset (SQUAD) [46] with a single task for BERT models and ALBERT-XX-L, (4) WikiText-2 [1] for GPT-2-L, and (5) CIFAR-10 [31] for ViT. The dimension (d) of $\mathcal{Q}$, $\mathcal{K}$, and $\mathcal{V}$ vectors for all the workloads is 64 except MemN2N with bAbI dataset, which is 20. The sequence length is 50 for MemN2N with bAbI whereas 512 and 384 for BERT and ALBERT-XX-L models with GLUE and SQUAD datasets, respectively. Finally, the sequence length for GPT-2 with WiKiText-2 is 1280.

Fine-tuning details. We use the baseline model checkpoints from HuggingFace [65] with PyTorch v1.10 [44] and fine-tune the models on an Nvidia RTX 3090, except for GPT-2-Large, for which we use an Nvidia A100. For default task-level training, we use the Adam optimizer with default parameters and the learning rate of $\{2, 3\} \times e^{-5}$ (same as baseline). To obtain the layer-specific threshold values, we perform an additional pruning-aware fine-tuning step for one to five more epochs to learn the optimal values while maintaining the baseline model accuracy. For this step, we use the learning rate of $1e^{-2}$ for $\mathcal{T}h$ ($5e^{-6}$ for the other parameters), as training for the $\mathcal{T}h$ is generally slower and a higher learning rate facilitates con-

vergence. To leverage faster fixed-point execution, we perform a final post-training quantization step with 12 bits for inputs in QK-PU hardware block and 16 bits for V-PU block similarly to [62].

Hardware design details. Table 3.1 lists the microarchitectural parameters of a single LeOPArd tile for two studied configurations: (1) A LeOPArd tile with six and (2) eight QK-DPUs that share a single 1-D MAC array in V-PU. The number of QK-DPUs is set such that the compute utilization for front-end and back-end units is balanced, while considering the pruning and bit-level early-termination rates across all the workloads. We synthesised and performed Placement-and-Route (P&R) for our designs with two tiles. The on-chip memory sizes for $\mathcal{K}$ and $\mathcal{V}$ are designed to store up to 512 sequences for a single head in a layer for both configurations.

Accelerator synthesis and simulations. We use Cadence Genus 19.1 [6] and Cadence Innovus 19.1 [7] to perform logic synthesis, floorplan, and P&R for the LeOPArd accelerator. We use TSMC 65 nm GP (General Purpose) standard cell library for the synthesis and layout generation of the digital logic blocks. These digital blocks are rigorously generated to meet the target frequency of 800MHz in consideration of all the CMOS corner variations and temperature conditions from $-40°$ to 125°C. For the SRAM on-chip memory blocks, we use Memory Compiler with ARM High density 65 nm GP 6-transistor based single-port SRAM version r0p0 [2].

We also develop a simulator to obtain the total cycle counts and number of accesses to memories for both LeOPArd and baseline accelerators. The simulator incorporates the pruning rate and the bit-level early-termination statistics for each individual workload. Using these statistics, the simulator evaluates runtime and total energy consumption of the accelerators.

Comparison to baseline architecture. We compare LeOPArd to a conventional baseline design without any of our optimizations (e.g. runtime pruning and bit-level early compute termination). For a fair comparison, we use the same frequency, bitwidths for $\mathcal{Q} \times \mathcal{K}^T$ and $\times\mathcal{V}$, and on-chip memory capacity for all the designs. The baseline design employs a single 12×12-bit QK-DPU as op-

posed to multiple 12×2-bit-serial ones, while both designs have the same back-end V-PU. As shown in Table 3.1, we evaluate LeOPArd under two design configurations. The first design with six QK-DPUs, dubbed Area-Efficient LeOPArd (AE-LeOPArd), almost perfectly matches the area of the baseline design (¡ 0.2% overhead) and provides an iso-area comparison setting. The second one with eight QK-DPUs, dubbed Highly-Parallel LeOPArd (HP-LeOPArd), provides an area 15% larger than baseline and delivers a better balance in the compute utilization of the front-end and back-end stages.

Comparison with A^3 and SpAtten. We also compare LeOPArd with two state-of-the-art attention accelerators, A^3 [20] and SpAtten [62], with support for runtime pruning. A^3 employs token pruning by comparing the Softmax output (probability) to a relative threshold, which is set using a user-defined parameter that adjusts the level of approximation. A^3 also employs a sorting mechanism to make the pruning decision after processing only a small number of large elements from the sorted $\mathcal{K}$ matrix in the order of magnitude. SpAtten performs cascaded head and token pruning by comparing the Softmax output with a user-defined threshold obtained empirically. There are no raw performance/energy results for individual workloads and simulation infrastructures of the accelerators. Therefore, we follow the comparison methodology of SpAtten [62], using throughput (GOPs / s), energy efficiency (GOPs / J), and area efficiency (GOPs / s / mm^2) metrics to provide the best comparisons. Both A^3 and SpAtten are implemented in 40 nm technology. To provide a fair comparison, we scale HP-LeOPArd from 65 nm to 40 nm based on both Dennard scaling (indicated with †) and measurement-based scaling rules [53] (indicated with ‡). We use a single tile with an area comparable to A^3 and SpAtten. Moreover, A^3 implements the $\mathcal{Q} \times \mathcal{K}^T$ using 9 bits as opposed to 12 bits in LeOPArd. As such, we scale the QK-PU of HP-LeOPArd from 12 bits to 9 bits to provide a head-to-head comparison with the A^3 accelerator.

2.4.2 Accuracy and Algorithmic Optimization

Impacts on model accuracy. Figure 3.9 compares the accuracies of the LeOPArd gradient-based on-the-fly pruning method and the baseline models in their vanilla implementation [65], across various tasks of evaluated workloads. On average, across all the evaluated tasks, LeOPArd runtime pruning degrades accuracy by only 0.07% for MemN2N with the bAbi dataset, 0.31% and 0.33% for BERT-B and BERT-L with the GLUE dataset, and 0.26% and 0.21% for BERT-B and BERT-L with the SQUAD dataset. For ALBERT-XX-L with the SQUAD dataset, the LeOPArd runtime pruning leads to only an 0.07% accuracy loss, whereas the degradation for ViT-B with the CIFAR-10 dataset is 0.76%.

In the GPT-2-L model, we use perplexity, which is the key metric for auto regressive language models. Note that perplexity is derived from the model loss, and thus lower perplexity is better. As shown in Figure 3.9-(f), LeOPArd runtime pruning results in a 0.07 decrease in perplexity. This is achievable because LeOPArd learns the optimal threshold values and co-adjusts them with the weight parameters simultaneously via gradient-based optimization. Figure 3.9 also illustrates that the LeOPArd pruning-aware fine-tuning pass evenly improves the accuracy for some of the benchmark tasks, with the maximum of 2.2%. However, this also degrades the accuracy for other tasks with the maximum of 2.6%. This accuracy fluctuations are unavoidable due to randomness in deep learning training, but overall the accuracy degradation, averaged across the evaluated benchmarks, converges adequately to a near-zero value ($\leq 0.2\%$). Performing the post-training quantization adds at most only 0.1%, for both the baseline and our pruning-aware fine-tuned models.

Runtime pruning rate analysis. Figure 2.8 shows the percentage of total $\mathcal{Q} \times \mathcal{K}^T$ *Score*s that are pruned away by our method using the learned threshold values across various benchmarks. In transformer software implementations, zeros are padded to maintain regular vector length despite the varying sequence length in each workload. The padded zeros are not counted for sparsity contribution in this paper. On average, LeOPArd prunes 91.7% (max. 97.4%) of *Score*s across all the 20 tasks for the MeMN2N model with the bAbI dataset. LeOPArd achieves

Table 2.3: LeOPArd performance comparison under different scenarios with prior work [20, 62].

Metric (unit)	A^3-Base	A^3-Conserv	SpAtten	HP-LeOPArd	HP-LeOPArd †	HP-LeOPArd ‡	HP-LeOPArd †*	HP-LeOPArd ‡*
Process (nm)	40	40	40	65	40	40	40	40
Area (mm^2)	2.08	2.08	1.55	3.47	1.31	1.31	1.05	1.05
Key Buffer (KB)	20	20	24	48	24	24	24	24
Value Buffer (KB)	20	20	24	64	24	24	24	24
($\mathcal{Q}$, $\mathcal{K}$)-bits	(9, 9)	(9, 9)	(12, 12)	(12, 12)	(12, 12)	(12, 12)	(9, 9)	(9, 9)
GOPs / s	259.0	518.0	728.4	574.1	932.8	1084.9	1143.9	1330.3
GOPs / J	2354.5	4709.1	772.9	519.3	2224.8	2028.8	3353.8	3058.4
GOPs / s / mm^2	124.5	249.0	470.0	165.5	710.4	826.1	1093.8	1272.1

† Dennard scaling trend applied to map on 40 nm process – ‡ Scaling rule from [53] applied to map on 40 nm process – *scaled to 9 bit $\mathcal{Q}$, $\mathcal{K}$

the average pruning rates of 78.6% (max. 93.2%) and 75.5% (max. 93.0%) for the BERT-B and BERT-L models with the GLUE dataset, while achieving 73.9% and 74.1% with the SQUAD dataset, respectively. Moreover, LeOPArd provides a 72.6% pruning rate for ALBERT-XX-L with the SQUAD dataset, 60.3% for ViT-B with the CIFAR-10 dataset, and 73.9% for GPT-2-L with the WikiText-2 dataset. As the results suggest, LeOPArd can significantly prune out the *Score*s across various tasks, with greater benefits to MeMN2N tasks compared to the BERT ones. We conjecture the lower pruning rates in BERT models are due to the higher probability of correlation between various tokens in the more complex language processing tasks compared to MemN2N.

As Figure 2.8 shows, in the case of ALBERT-XX-L with SQUAD, we see more pruning opportunities compared to BERT, presumably because of its larger model architecture with more redundant computations. Similar trend is observed for GPT-2-L. With regard to ViT-B, we see lower pruning compared to NLP tasks, commensurate with prior studies [11]. This occurs because information is more local in images compared to texts, and therefore there is less redundancy in the attention layers for vision tasks.

Bit-level early-compute termination. Figure 2.9 depicts the proposed bit-level early compute termination feature and its relation with the achieved runtime pruning rates. The x-axis shows the number of bits processed sequentially, while the y-axis shows the cumulative achieved pruning rate averaged over all of the datasets' tasks. Intuitively, as more bits are processed during *Score* computations, the dynamic margin becomes smaller and thus the pruning rate increases. As shown, as the average number of processed bits increases, the cumulative prun-

ing rate gradually plateaus, indicating saturation. In this scenario, the higher number of bits are only required for fully calculating unpruned *Score*s. We establish that the lower redundancy in model parameters of some transformer models, e.g. BERT-L / ViT-B, hinders higher runtime pruning. Because lower redundancy generally translates to a higher number of average bits calculations, it proportionally diminishes the potential gains from bit-wise early termination. Averaged over pruned *Score*s in bit-serial mode, MemN2N with the bAbi dataset requires 4.5 bits, while BERT-B and BERT-L require 8.3 and 8.0 bits with the GLUE dataset. With the SQUAD dataset, the average number of bits in BERT-B and BERT-L are 7.6 and 9.0 bits, whereas ALBERT-XX-L maintains 8.0 bits. The average number of bits in GPT-2-L and ViT attain 7.6 bits and 8.5 bits, respectively. This devised early-termination mechanism significantly reduces the computations of the $\mathcal{Q} \times \mathcal{K}^T$.

2.4.3 Accelerator Performance Results

Performance and energy comparison to baseline. Figure 3.11 shows the speedup improvements delivered by LeOPArd compared to the baseline design, across all the 43 studied tasks. In this comparison, we consider the total execution runtime for all attention layers of the models. On average across all tasks, AE-LeOPArd and HP-LeOPArd provide 1.9× and 2.4× speedup over the baseline, respectively. These improvements stem from both LeOPArd runtime pruning that reduces operations on the back-end unit (e.g., Softmax and $\times\mathcal{V}$) and bit-level early compute termination that saves cycles on $\mathcal{Q} \times \mathcal{K}^T$ computations for pruned *Score*s. Across the workloads, LeOPArd delivers higher speedups for MemN2N compared to the other benchmarks. We attribute these improvements to the higher pruning rate and consequently more bit-level termination opportunities in this model's tasks. Among all the tasks, MemN2N-Task-1 enjoys the maximal speedup (3.8× for AE-LeOPArd and 5.1× for HP-LeOPArd) while ViT-B gains the minimal improvements (1.1× for both AE-LeOPArd and HP-LeOPArd). The benefits are more pronounced for HP-LeOPArd because it deploys more QK-DPUs, which both improves the performance of the front-end Q-PU unit, and

delivers more inputs (*Score*s) to the back-end stage. The latter generally increases the back-end utilization.

Figure 3.12 compares the energy reduction (including compute and on-chip memory accesses) achieved by LeOPArd to the baseline. On average, LeOPArd reduces total energy consumption by a factor of 3.9× for AE-LeOPArd and 4.0× for HP-LeOPArd, across all the studied tasks. Similarly to the speedup comparisons, MemN2N enjoys a greater energy reduction than the other benchmarks due to the higher pruning rate and therefore faster bit-level compute terminations. Across all tasks, the energy reduction is the greatest for MemN2N-Task-1 (9.2× for AE-LeOPArd and 9.6× for HP-LeOPArd) and ViT-B achieves the lowest savings (≈2.0× for AE-LeOPArd and HP-LeOPArd). The impact of LeOPArd on energy exceeds that on speedup, because runtime pruning and bit-level early termination reduce computation energy (contributing to both energy savings and speedup) and memory accesses (only contributing to energy savings). The energy reductions for both AE-LeOPArd and HP-LeOPArd are not substantially different. Because the additional QK-DPUs in HP-LeOPArd increase both power and performance, total energy consumption remains similar.

Analysis of energy savings breakdown. Figure 3.13 analyzes the breakdown of total energy consumption across five microarchitectural components: (1) $\mathcal{Q} \times \mathcal{K}^T$ computations, (2) $\mathcal{K}$ buffer memory access, (3) Softmax, (4) $\times\mathcal{V}$ computations, and (5) value buffer memory access. We report the average breakdown across all tasks for each workload. Additionally, Figure 3.13 illustrates the contribution of LeOPArd's two main optimizations: (1) runtime pruning and (2) early compute termination through bit-serial execution to the overall energy savings in AE-LeOPArd. We normalize the energy breakdowns to a baseline, which does not utilize any of the LeOPArd's optimizations. In the baseline, $\times\mathcal{V}$ computations and value buffer memory accesses proportionally consume the highest energy due to the lack of runtime pruning; ergo, higher average number of bits in $\mathcal{Q} \times \mathcal{K}^T$. Recall that the LeOPArd's back-end unit encloses Softmax, $\times\mathcal{V}$, and its associated buffer accesses. As the results show, this unit consumes more than 65% of the total energy in the baseline design. LeOPArd's runtime pruning enables

skipping computations and memory accesses for inconsequential *Score*s during the back-end processing, delivering 1.7× (ViT-B) to 2.5× (MemN2N) energy savings. For these tasks, the bit-serial execution in LeOPArd along with its early termination brings further energy savings of 1.3× (ViT-B) to 2.3× (MemN2N) on top of runtime pruning. These additional benefits arise from avoiding the inconsequential bit computations in $\mathcal{Q} \times \mathcal{K}$ and their associated $\mathcal{K}$ buffer accesses.

Comparison with A^3 and SpAtten. Table 3.3 compares the characteristics and performance of HP-LeOPArd and its scaled versions with A^3 and SpAtten. Compared to SpAtten, HP-LeOPArd† (HP-LeOPArd‡) delivers 3× (2.6×) improvements in GOPs / J and 1.5× (1.7 ×) improvements in GOPs / s / mm^2, while both designs have virtually no model accuracy degradation. These benefits are attributed to the LeOPArd's higher pruning rate and to the bit-level early compute termination. For comparison with A^3, we evaluate HP-LeOPArd$^{\dagger *}$ (HP-LeOPArd$^{\ddagger *}$), which are scaled to 40 nm and deploy 9-bit arithmetic for $\mathcal{Q} \times \mathcal{K}^T$. A^3-Conservative deploys heuristic approximation to minimize accuracy degradation on top of A^3-Base, which does not use approximation. HP-LeOPArd$^{\dagger *}$ (HP-LeOPArd$^{\ddagger *}$) achieves 1.4× (1.3×) higher energy efficiency (in GOPs / J) and 8.8× (10.2×) area efficiency (in GOPs / s / mm^2) than A^3-base. HP-LeOPArd$^{\dagger *}$ (HP-LeOPArd$^{\ddagger *}$) also provides 4.4× (5.1×) improvements in terms of GOPs / s / mm^2 compared to A^3-Conservative. Although A^3-Conservative provides 29% and 35% higher energy efficiency compared to HP-LeOPArd$^{\dagger *}$ and HP-LeOPArd$^{\ddagger *}$, respectively, this comes at the cost of visible accuracy degradation, e.g., 1.0% for MemN2N and 1.3% for BERT-Base with the SQUAD dataset as reported in [20]. On the other hand, LeOPArd's carefully crafted gradient-based training balances pruning rate and model accuracy, providing accuracy degradation of only 0.06% and 0.26% for the aforementioned models and datasets without manual configurations for heuristic parameters.

LeOPArd accelerator layout area details. Figure 3.14(a) shows the layout of LeOPArd architecture, which occupies 2.3 × 2.8 mm^2, including two tiles. The layouts are generated by meeting the design rule check in a 65 nm process and targeting 65-75% physical density, commonly used for the routing convenience and

tape-out yield. Figure 3.14-(b) reports the area breakdown, where QK-DPU takes the largest proportion as we employ N_{QK} QK-DPU in consideration of the high pruning rate. This leads to 56% area occupied by the front-end unit, which includes QK-DPU and $\mathcal{K}$ buffer. The on-chip memory for $\mathcal{K}$ and $\mathcal{V}$ occupies 34% of the layout area.

2.4.4 Architecture Design Space Exploration

QK-PU parallelism degree. As discussed in Section 2.3.1, the number of QK-DPUs (N_{QK}) within one QK-PU exhibits a trade-off space in designing the LeOPArd accelerator. To find the number of QK-DPUs that balances the utilization of front-end and back-end units, we sweep the N_{QK} from three to 12 in Figure 2.14 and report the V-PU utilization across the evaluated tasks. If utilization exceeds 100% (common when $N_{\text{QK}} = 12$), the back-end V-PU is over-subscribed due to the throughput mismatch between V-PU and QK-PU. This mismatch throttles the back-end V-PU and turns into the system bottleneck, frequently stalling the front-end. On the other hand, when $N_{\text{QK}} = 3$, the V-PU is chronically underutilized due to a significant reduction in its number of computations, attributed to front-end runtime pruning mechanism. As marked by dark green diamonds, $N_{\text{QK}} = 8$ adequately balances the V-PU utilization and the number of front-end unit stalls. Thus, we favor this configuration for HP-LeOPArd. The second best configuration to balance front- and back-end utilization is $N_{\text{QK}} = 6$ (marked by light green diamonds). As such, we choose this configuration for AE-LeOPArd, which matches the baseline chip area usage.

Bit-serial processing granularity. Figure 2.15 illustrates the design space exploration for granularity of the bit-serial execution in QK-DPU ($\mathcal{B}$). This bit-level granularity creates a trade-off space, where decreasing the $\mathcal{B}$ stores intermediate results at the end of each bit processing cycle more frequently (escalating the energy). At the same time, increasing $\mathcal{B}$ curtails the performance of early compute termination due to lower resolution in stopping the computations. To find the optimal point, we sweep the $\mathcal{B}$ for values of 1, 2, 4, and 12 bits and measure the average consumed energy and its breakdown ($\mathcal{Q} \times \mathcal{K}^T$ logic and key buffer ac-

cesses) per one output *Score*. All the numbers are normalized to 12-bit processing that does not employ any bit-serial execution. Figure 2.15 depicts this analysis for MemN2N tasks (results for other models are similar) and reports the average across all tasks. As shown, 2-bit-serial execution strikes the right balance between energy consumption of the bit-serial computations and the resolution of bit-level early compute termination.

Chapter 2 is adapted from the material as it appears in Zheng Li, Soroush Ghodrati, Amir Yazdanbakhsh, Hadi Esmaeilzadeh, Mingu Kang. 2022. Accelerating Attention through Gradient-Based Learned Run- time Pruning. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. The thesis author was the primary investigator and author of this paper.

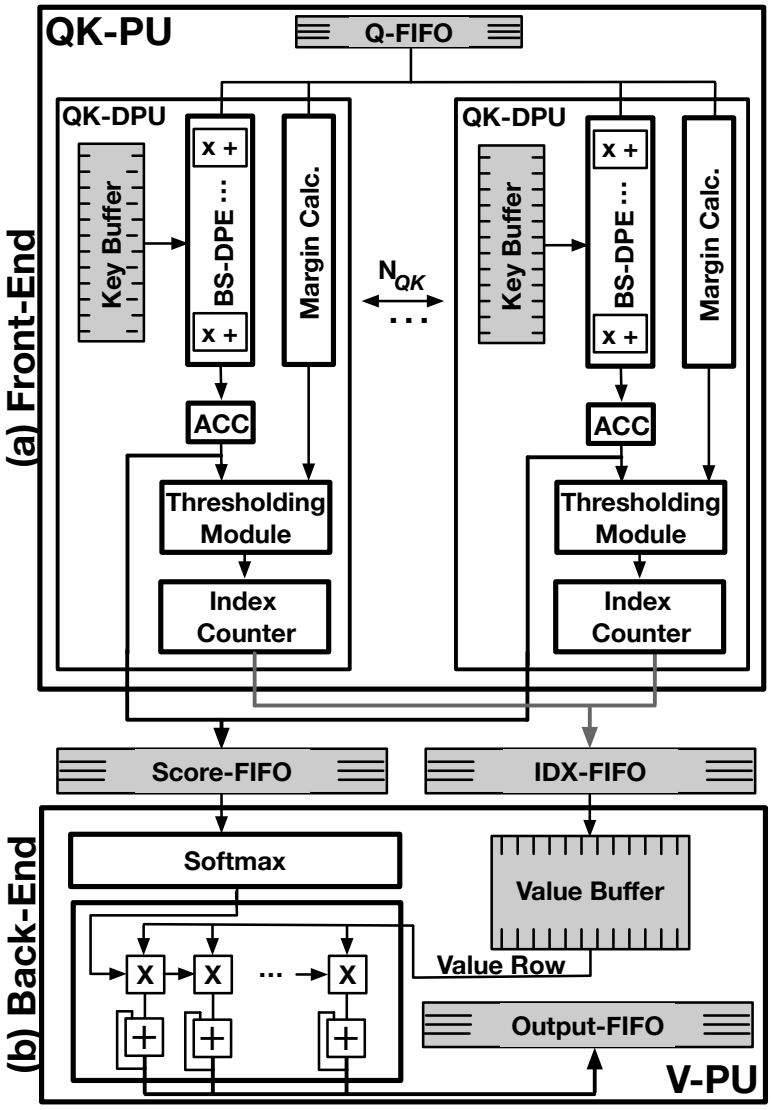


Figure 2.5: Overall microarchitecture of a LeOPArd tile.

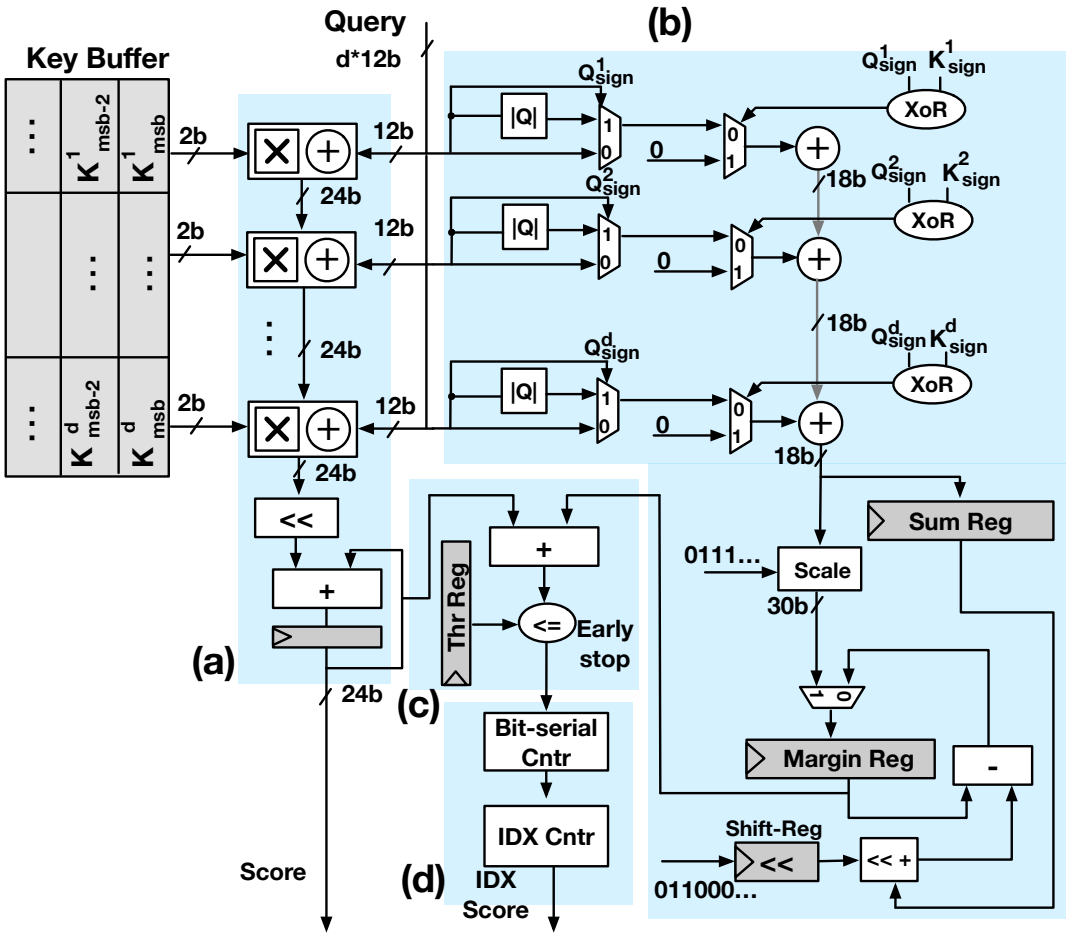


Figure 2.6: A QK-DPU comprising (a) bit-serial dot-product engine, (b) margin calculation logic, (c) thresholding module, and (d) score index counter.

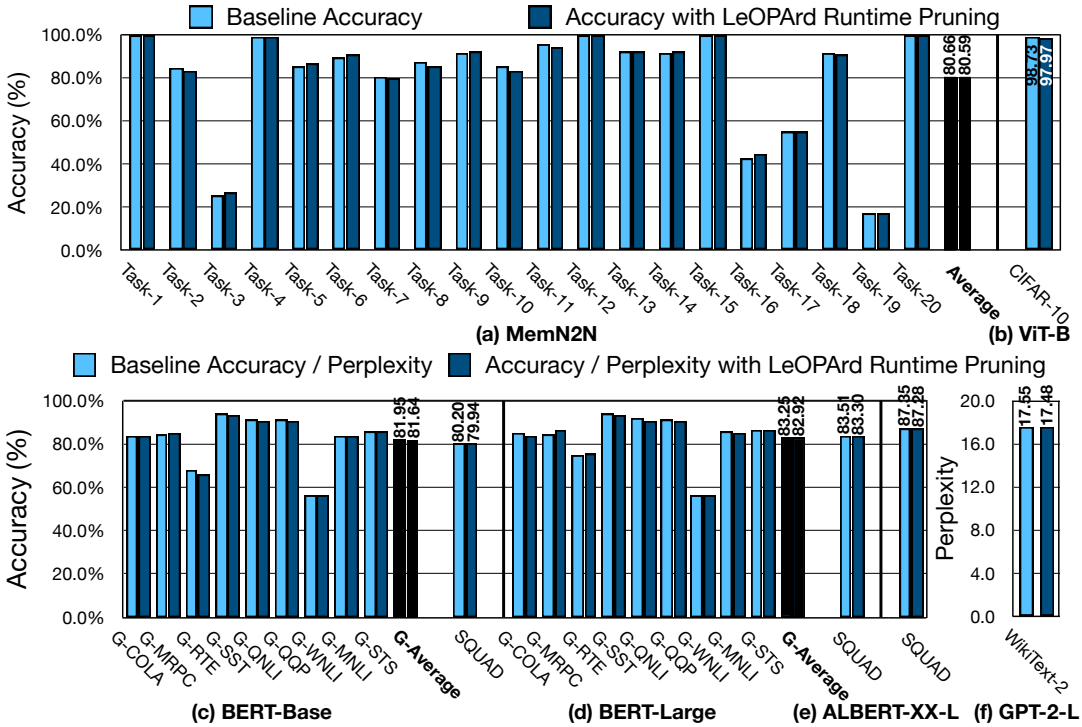


Figure 2.7: Accuracy before and after pruning-aware fine-tuning (prefix "G-": GLUE). We evaluate GPT-2 using perplexity, which favors a lower value.

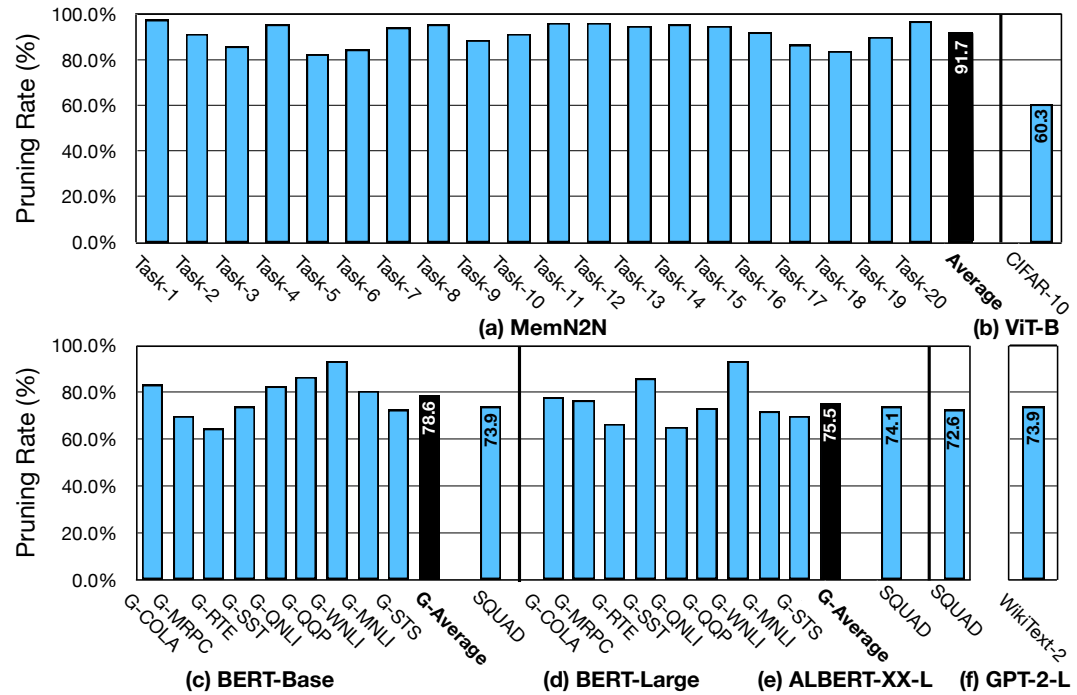


Figure 2.8: Runtime pruning rate with LeOPArd. (prefix "G-": GLUE)

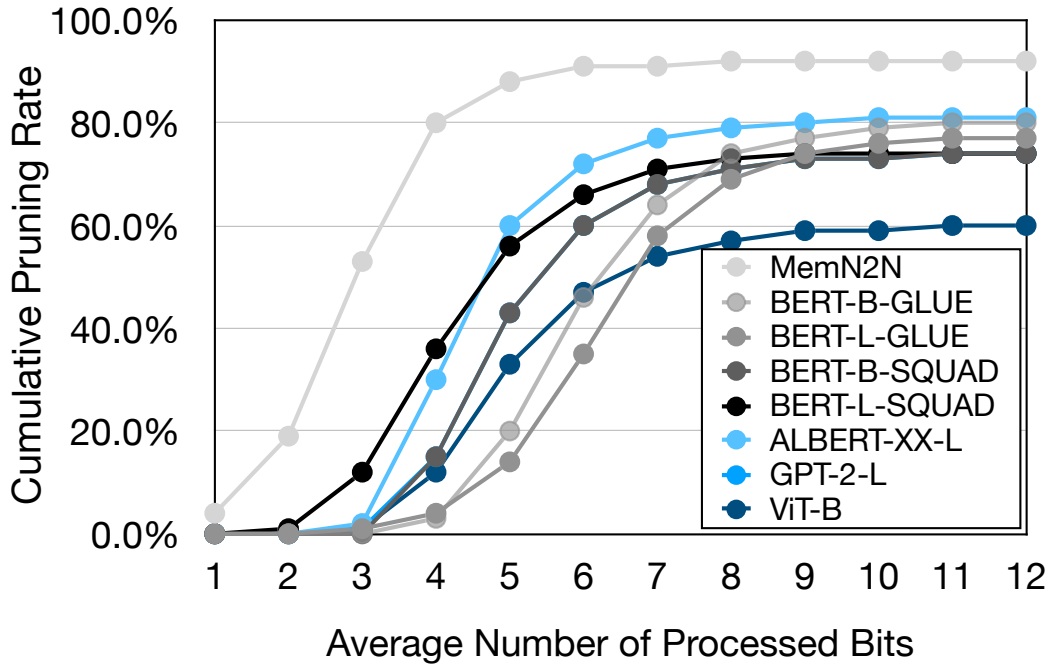


Figure 2.9: Cumulative pruning rate with respect to the number of bits processed during bit-serial early termination. Each line obtained by averaging across all the pruning rates per task.

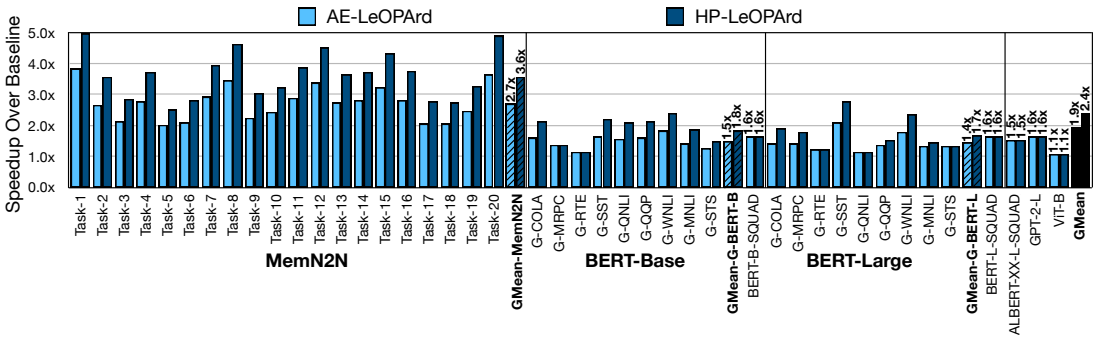


Figure 2.10: Speedup comparison to baseline design for AE-LeOPArd and HP-LeOPArd (prefix "G-": GLUE dataset).

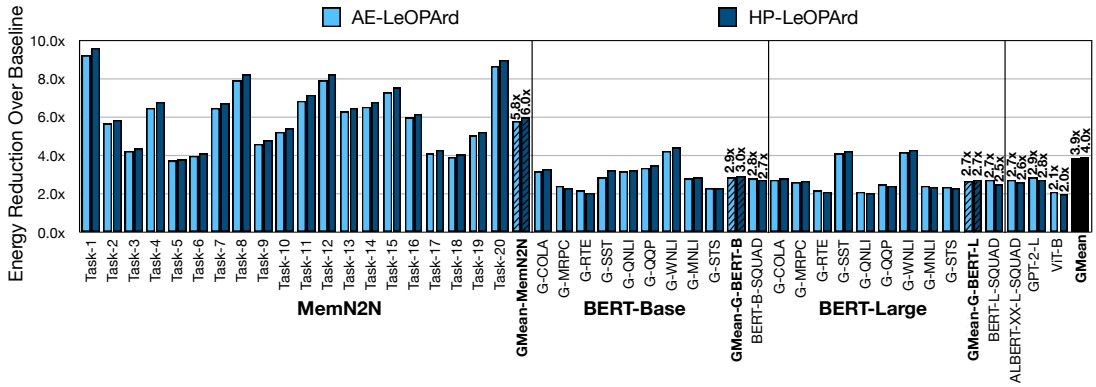


Figure 2.11: Total energy reduction for AE-LeOPArd and HP-LeOPArd compared to baseline (prefix "G-": GLUE dataset).

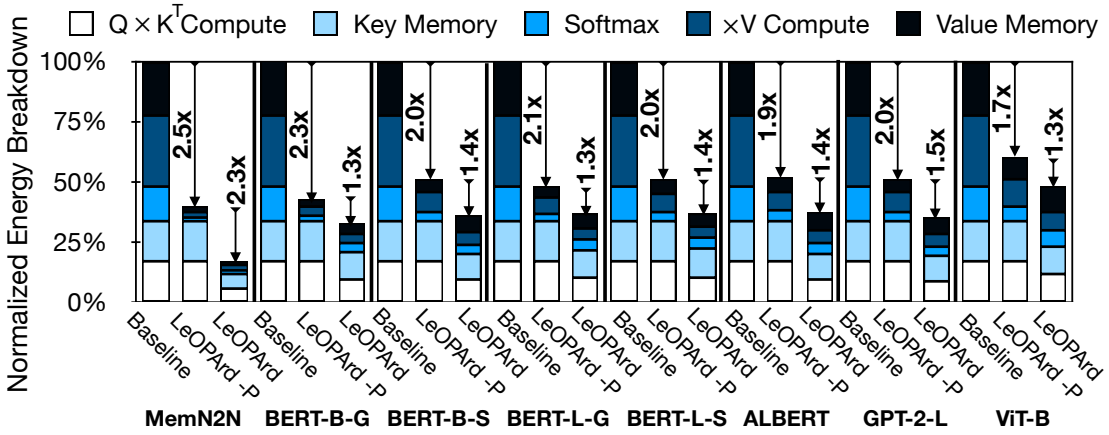
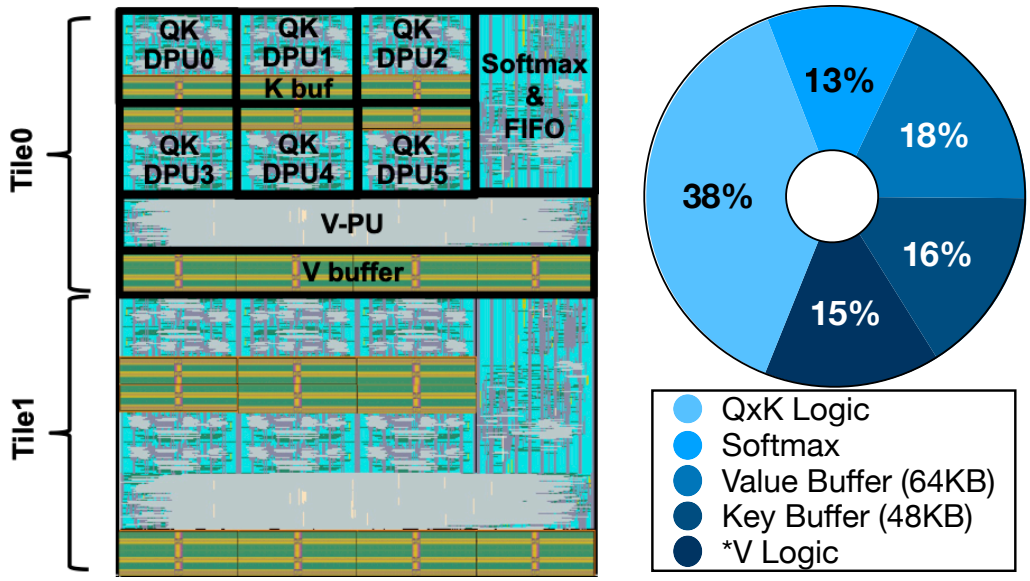


Figure 2.12: Normalized LeOPArd's average energy breakdown and the contribution of runtime pruning and bit-level early termination in energy saving (-P: only pruning, LeOPArd: pruning + bit-serial early termination)



(a) (b)

Figure 2.13: AE-LeOPArd: (a) layout (2.3×2.8 mm^2) and (b) area breakdown.

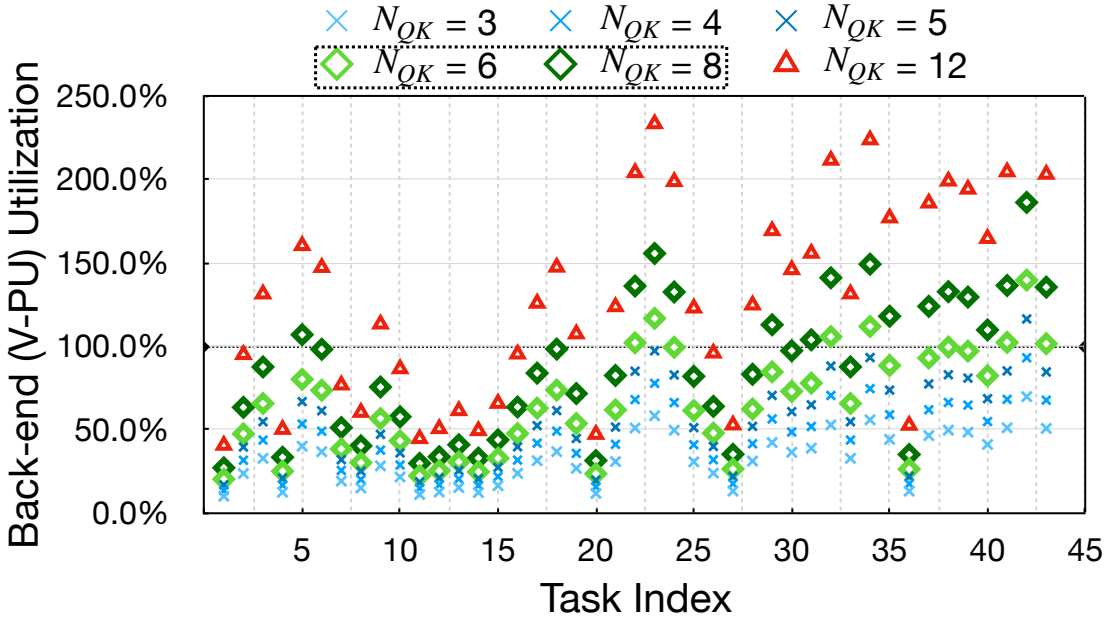


Figure 2.14: Back-end V-PU utilization over the QK-PU parallelism (N_{QK}). $N_{QK} = 6$ and $N_{QK} = 8$ form the favorable configurations in terms of back-end utilization in AE-LeOPArd and HP-LeOPArd, respectively.

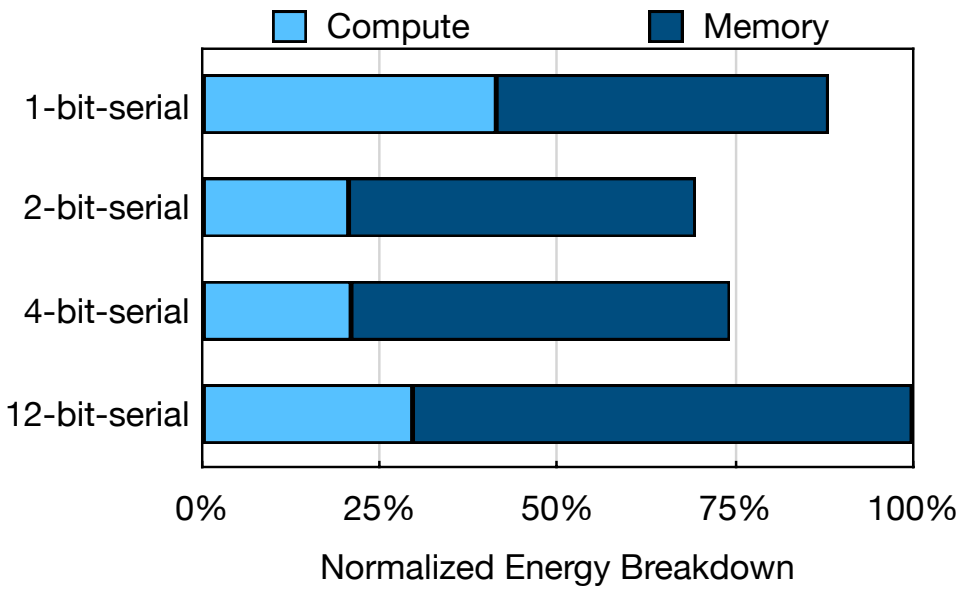


Figure 2.15: Design space exploration for the resolution ($\mathcal{B}$) of bit-serial execution with respect to normalized average QK-DPU energy per *Score*.

Chapter 3

Sparse Attention Acceleration with Approximate In-Memory Pruning

In Chapter 2, it is assumed that the whole input sequence can be loaded to the on-chip memory. However, as the input sequence become longer, this assumption may not hold in many edge devices anymore. This Chapter proposes a solution to reduce data movement transaction when the on-chip memory is limited.

3.1 Motivation

The LeOPArd design in Chapter 2 features efficient pruning, however, the considerable overhead of data communication even with adequately sized on-chip buffers is a problem remains to be solved. The data communication cost is exacerbated when on-chip resources are limited, because of frequent instances of data communication. Figure 3.1 measures the contribution of off-chip memory read and write accesses to the overall energy consumption to process a single-head self-attention layer[1]. The x-axis encompasses various fractions of on-chip memory capacity with respect to different input sequence lengths.

[1]Section 3.6 outlines the experimental setup details.

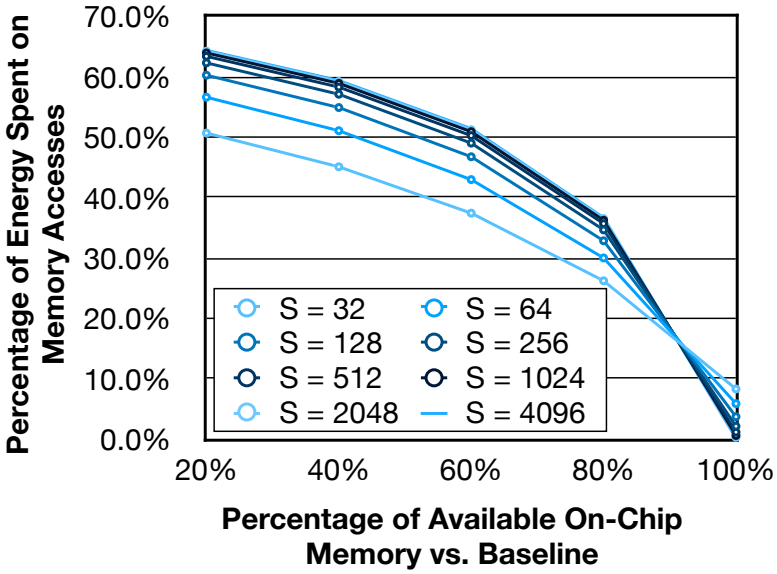


Figure 3.1: Percentage of energy spent on memory accesses to process one attention head with respect to various percentages of available on-chip memory. The results are shown across various sequence ($\mathcal{S}$) length.

As on-chip resources become scarce (20% of requisite on-chip buffers available to store the entire key and value matrices), on average, the energy contribution of on-chip memory increases to $>60\%$, turning into the dominant energy contributor. In this tightly-budgeted scenario, approaches that unlock the opportunity to fetch only a subset of relevant data become attractive.

One such compelling solution is applying the run-time pruning introduced in Chapter 2. However, pruning require to bring in the entire key and value matrices to exercise thresholding. As discussed in Chapter 1, the data movement can be greatly reduced if the pruning operation is done on the main memory. Then the data needed to be fetched from the main memory to the processor will be greatly reduced. This chapter shows solutions that tackles above challenge by approximating $\mathcal{Q}\times\mathcal{K}^T$, followed by a comparison with threshold values. Despite the approximation, our results ensure that this in-memory thresholding mechanism can consistently identify the entire subset of relevant vectors. To guarantee accuracy on par with baseline, we recompute the score values in a precise manner after selective data fetching.

3.2 Data Communication Optimization

In-memory Thresholding

Under scarce on-chip resources, a logical optimization step can leverage in memory computing to eliminate inconsequential data communications for pruned key and value vectors. For example, in Figure 3.2, the core simply stipulates $\mathcal{K}_{2,4,5,6,11,13}$ for $q_1 \times \mathcal{K}^T$ computations ($q_1 \rightarrow$The). This observation provides the opportunity to significantly cut costs by informing the accelerator to only fetch the requisite data.

Spatial Locality in Adjacent Queries

While in-memory thresholding trims down the amount of data per query that are brought into on-chip buffers, it increases the frequency of data fetches. This is because a new set of key and value vectors should be fetched to proceed computing for subsequent queries once the computations for $q_i \times \mathcal{K}^T$ completes. This increase in the frequency of data fetches may well nullify the potential benefits of reducing the amount of transferred data.

To explore future potential reductions in the amount of transferred data and compensate for the likely overhead of frequent data transfers, we study the similarities between unpruned keys across input queries. Figure 3.2 illustrates a real example of CoLA task from GLUE dataset [61] (eighth head in the first attention layer). Each row indicates a query and its corresponding unpruned key locations, filled in green. The grey shading on the last few rows and columns specifies the input mask, commonly used in transformer models when the sequence length in the input dataset is less than the one in the model. It is visually evident that a significant number of keys are inconsequential per query, and that there is a high spatial locality between adjacent rows. For example, compared to query The, the additional required keys for the adjacent query more are only appear and in. The remaining unpruned key elements, such as more, of, and him, are identical between these queries, obviating additional data transfers.

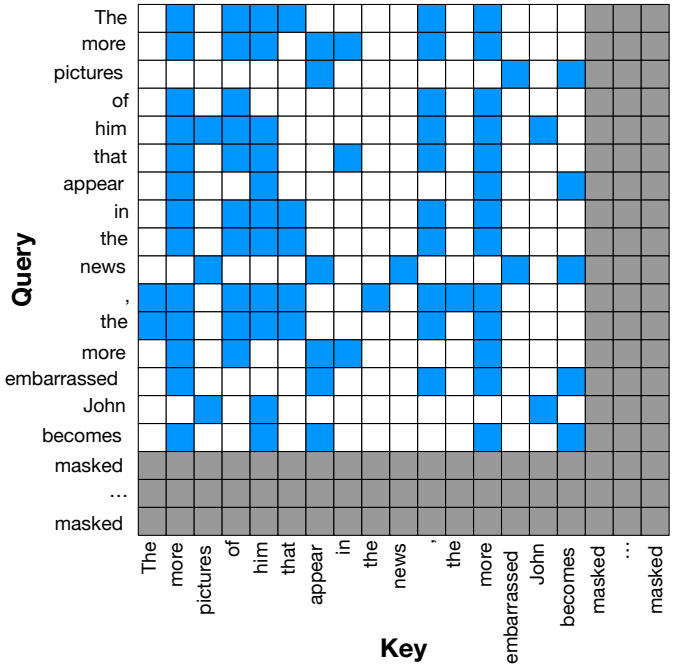


Figure 3.2: Query-Key relation for the first attention layer of CoLA task from GLUE dataset [61]. White squares represent pruned entries. The gray stripes are masked regions.

3.2.1 Theoretical expectation of spatial locality.

Equation 3.1 calculates the probability of $\mathcal{L}$, defined as the number of overlapping elements between adjacent queries of size $\mathcal{S}$. In this equation, $\mathcal{M}$ represents the number of the unpruned elements in each query. The probability of $\mathcal{L}$ is calculated by first multiplying the numbers of possible combinations of $\mathcal{L}$ elements out of $\mathcal{M}$ and the remaining $\mathcal{M}$ - $\mathcal{L}$ elements out of $\mathcal{S}$ - $\mathcal{M}$. This product is subsequently divided by the number of possible combinations of $\mathcal{M}$ elements out of $\mathcal{S}$. The resulting probability of each $\mathcal{L}$ is then multiplied by the value of $\mathcal{L}$ and summed across $\mathcal{M}$ to calculate the theoretical expected overlap between adjacent queries, as demonstrated in Equation 3.1.

$$P(\mathcal{L}) = \frac{[\mathcal{M}]\mathcal{L} \times [\mathcal{S} - \mathcal{M}]\mathcal{M} - \mathcal{L}}{[\mathcal{S}]\mathcal{M}} \quad E(\mathcal{L}) = \sum_{\mathcal{L}=1}^{\mathcal{M}} \mathcal{L} \cdot P(\mathcal{L}) \tag{3.1}$$

In Figure 3.3, we compare the percentages of overlaps, averaged across multiple inputs and observed in various extant datasets [27, 15, 45], with the theoretical

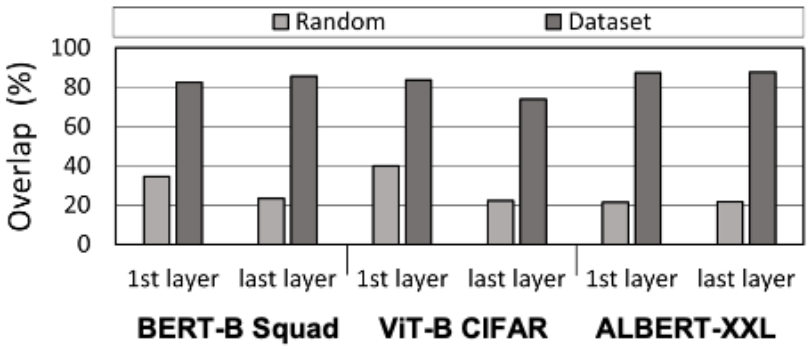


Figure 3.3: Number of common indices between neighboring tokens ($\mathcal{Q}_i$ vs. $\mathcal{Q}_{i+1}$) with the practical dataset vs. randomly selected pruned tokens with the pruning rate from [37].

expectation formula, as presented in Equation 3.1. The results reveals a striking 2 - 3× increase in the observed overlap percentage in the real world scenarios. This increase highlights a notable data reuse opportunity because most of the requisite elements already reside in on-chip buffers. Therefore, exploiting this data reuse opportunity limits the number of data fetches only to the unpruned elements that differ between adjacent queries, leading to a dramatic cost reduction.

Futile Computations in Padded Regions

It is a common practice [65] in transformer models to pad input sequences that are shorter than the maximum supported length. The padded inputs do not meaningfully contribute to the self-attention computations, and hence are irrelevant for the final model accuracy. These padded regions are highlighted as gray squares in Figure 3.2, where only 16 queries out of 128 are computationally relevant. This leaves (128-16)×(128-16) score computations inconsequential. The padded regions are commonly nullified by placing a sufficiently large negative value [65]. Passing these negative values through Softmax prompts their probability to approach zero, excluding them from subsequent computations. To further eliminate unnecessary data communications in these padded regions, we can proactively identify them as early as possible in memory.

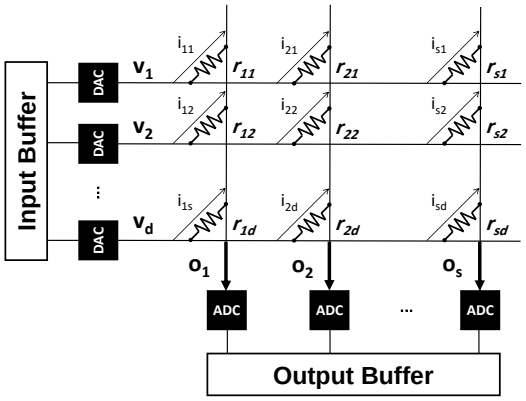


Figure 3.4: In-memory computing with ReRAM cross-bar array.

3.3 In-memory Thresholding

3.3.1 Overview of ReRAM.

Resistive Random Access Memory (ReRAM) is a non-volatile memory that stores data using its adjustable resistance. Figure 3.4 demonstrates a ReRAM 2D crossbar array [43]. To further improve the density and energy efficiency of ReRAM, recent methods [70, 66, 73, 39] use Multi-Level Cells (MLC) to store multiple bits of information inside each cell. In contrast to Single-Level Cells (SLC), the MLC ReRAM permits a range of resistance values inside each cell. Although storing more bits per cell appeals by increasing ReRAM memory density, it can easily become a limiting factor. As the number of bits/cell increases, each cell renders itself more amenable to circuit noises and limits the accuracy of computations. Recent studies [23, 4] deem a four bits/cell MLC ReRAM design the optimal balance between robustness and complexity of current sensing detection circuitry.

3.3.2 Vector-Matrix multiplication with ReRAM in-memory computing.

ReRAM can perform efficient and highly parallel analog vector-matrix multiplications, as demonstrated by prior work [49, 12, 48, 19] on DNN acceleration.

To perform such multiplications, the matrix elements are mapped onto memristor conductance and the input vector is fed into ReRAM's wordlines (Figure 3.4, horizontal lines), one element per row, as biased voltages. Additionally, a sum reduction can be executed on the resulting multiplications across the crossbar columns as serial currents [68, 36]. Once complete, the weighted-sum vector forms an analog current at the boundary of the ReRAM crossbar, one element per column. The following equation formally presents a multiplication between vector $v_{1\times n}$ and matrix $M_{n\times m}$ on a ReRAM crossbar array:

$$m_{ij} = \frac{1}{r_{ij}} \quad o_{1j} = \sum_{i=1}^{n} v_{1i} \cdot m_{ij} \tag{3.2}$$

where m_{ij} and r_{ij} represent each element of matrix M and its corresponding resistance value in ReRAM cells.

3.3.3 Application in run-time pruning.

The in-memory principle introduced above can be seamlessly applied for accelerating the attention mechanism. This can be achieved by storing each k_i vector in a column of the crossbar array, and applying the input voltage level, which corresponds to the element of query vector q_i, to each wordline as described in Figure 3.6(a). Ideally, we require s columns to store entire sequence length while d rows are needed to accommodate the entire embedding size. If the array size does not match with problem size, multiple banks of array can be employed in a tiled manner. All of k_i vectors stored in multiple columns are processed for parallel dot-product operations in one shot. Once it completes, the next query vector q_{i+1} is processed in the subsequent cycle.

3.3.4 Analog↔Digital challenges.

It is shown [13, 48, 36] that digital↔analog conversion drains a significant portion of total ReRAM power consumption, especially as the number of conversion bits increases. For example, the power and area of a 5-bit ADC are >20× [57] and

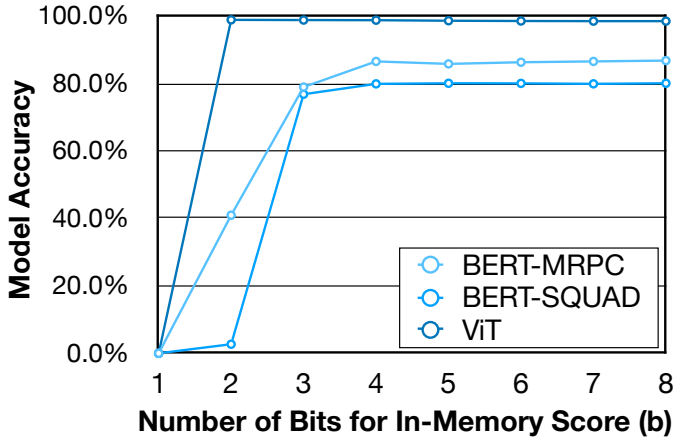


Figure 3.5: Sensitivity of model accuracy to the number of bits (b) used for in-memory thresholding (comparison of in-memory scores with $\mathcal{T}h$, Equation 3.3).

$>30\times$ [54] higher than a 1-bit ADC, respectively. Therefore, it is crucial to take the power overhead of these converters into account, especially for designs with greater than one-bit precision requirements. In the following, we discuss the main challenges to in-memory thresholding.

3.3.5 In-Memory Thresholding Challenges

1. **Analog computing inaccuracies.** Analog computing in ReRAM is commonly known to be susceptible to inherent circuit noises and inaccuracies, such as thermal noise and temperature fluctuations [22, 24], and coupling noise between adjacent cells [9]. These inaccuracies limit the feasible precision of computations in ReRAM crossbar arrays. To evaluate the impact of limited compute precision for in-memory thresholding on the final model accuracy, we use the following approach[2]:

$$\mathcal{P}rune = \text{Argwhere } (\text{Score}_{\text{R}}^{\text{b}} < \mathcal{T}h);\ \ \text{Score}[\mathcal{P}] = -c \tag{3.3}$$

where $\text{Score}_{\text{R}}^{\text{b}}$ denotes the in-memory score values (e.g. results of $q_i \times \mathcal{K}^T$) when the output has limited accuracy with a b-bit precision. "Argwhere" finds the indices of score elements that are lower than the target threshold.

[2]As we explain in Section 3.6, we do *not* perform additional fine-tuning to quantize key values to lower bit-precision.

Note that the threshold values ($\mathcal{T}h$) are learned during the full-precision finetuning process such as LeOPArd [37, 30]. The scores of the identified pruning indices are then forcefully set to a large negative value ($-c$) to remove irrelevant elements. Also, recall that we perform low-precision in-memory computing for the sole purpose of identifying the irrelevant key vectors. With on-chip accelerators, the score computation for unpruned vectors is still performed in full-precision.

Figure 3.5 compares the final model accuracy after quantizing the Score with different bit-precisions (b) across three different models: BERT-Base [27] with GLUE [61], BERT-Base with SQUAD [46], and ViT [15] with [31] dataset. The results show that the quantization error with 4-bit precision virtually has no impact on the final model accuracy[3]. Thus, the runtime pruning mechanism is robust against approximation, even when the computation has a certain level of errors. This is intuitive because the incorrectly pruned vectors already exhibit a small score value, likely in the vicinity of $\mathcal{T}h$. Hence, the impact on model accuracy is negligible. Finally, even more sensitive workload to the noise can be in theory compensated by adding a small margin on top of $\mathcal{T}h$ in Equation 3.3 at the cost of reducing the pruning ratio (directly proportional to hardware performance).

2. **ADC converter overhead.** The overhead of ADC converters increases proportionately to the precision of conversion. Two design choices can support comparisons between vector-matrix multiplication outputs and the threshold values. The first option uses a 5-bit ADC to convert the outputs and employs digital comparators for thresholding. The other option utilizes analog comparators for thresholding prior to ADC. The output of each analog comparison represents a binary value, which indicates whether to prune the corresponding key vector. Since the resulting pruning vector only requires one bit per key, we can use a low-overhead 1-bit ADC (implemented as a comparator). The low overhead of 1-bit ADC (>20× [57] lower area and

[3]A recent study from HP Lab [23] has shown that ReRAM in-memory computing for 64-tap dot-product delivers 5-bit equivalent output accuracy after including all the error sources.

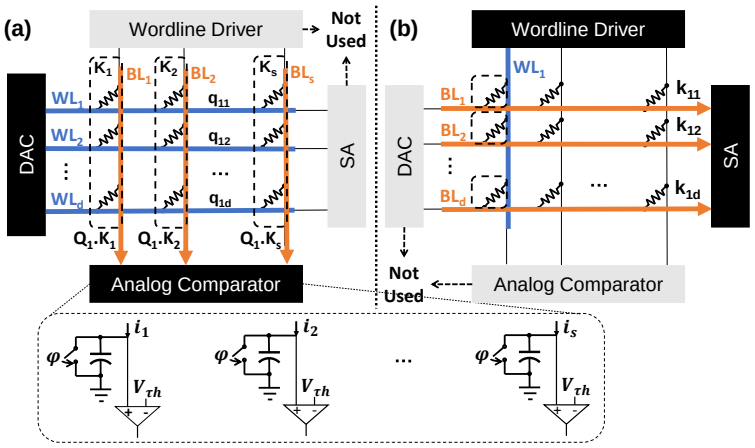


Figure 3.6: Transposable ReRAM crossbar array. (a) ReRAM crossbar during in-memory pruning, (b) Transposed ReRAM crossbar during normal read.

>30× [54] lower power consumption compared to a 5-bit ADC) favors the second option for in-memory thresholding.

3. **Reading unpruned vectors overhead.** Finally, performing in-memory thresholding followed by fetching each unpruned $\mathcal{K}$ vector from ReRAM arrays (for digital re-compute) is arduous and can impose significant read latency. This occurs because we store each vector of $\mathcal{K}$ vertically at each ReRAM column (Figure 3.4), and k_i is mapped to the i^{th} ReRAM column. On the other hand, accessing from ReRAM through a standard read operations fetches the data stored horizontally in a row. Therefore, fetching from ReRAM requires sequentially asserting *all* the (horizontal) wordlines, bringing in each row of the $\mathcal{K}$ matrix (even the ones associated with *pruned k vectors*), and selectively fetching the unpruned vectors to on-chip buffers. We address this challenge by a recent taped-out transposable ReRAM proposal [60], which we expound below.

3.3.6 Transposable ReRAM for Thresholding

Overview. A transposable ReRAM [60] supports (1) in-situ access to the array to perform vector-matrix computations (in-situ computation), as well as (2) reading *their transposed* values (transposed read). Figure 3.6 shows the overall de-

sign of a transposable ReRAM in these two modes. In the "in-situ computation" mode, the ReRAM array performs vector-matrix multiplications, similarly to conventional (non-transposable) ReRAM crossbar shown in Figure 3.4. In this case, we assign the value of each element in input vector q_i to wordlines (horizontal) and assert all the bitlines (vertical) to enable parallel multiplications. On the other hand, in the new "transposed read" mode, the horizontal lines become bitlines and vertical line becomes wordline. In this mode, *only* one wordline gets asserted. Once the bitline current from all the columns are fully developed, the sense amplifier reads all the values stored on the ReRAM conductance of the asserted wordline (in the column).

In-memory thresholding dataflow. As discussed in the previous section, one of the challenges for performing in-memory thresholding is reading unpruned vectors after score calculation. The "transposed read" mode presents a viable solution to this challenge. Next, we present a dataflow to identify unpruned key vectors leveraging transposable ReRAMs. In this dataflow, we store each key vector vertically in the ReRAM crossbar array (the first key vector is mapped onto the first ReRAM column, and so on). Because analog circuit noises limit the supported bit-precision on each memory cell, we only store a predefined subset of MSB-side bits within each cell. Our experiment showed that a 4-bit precision is sufficient for in-memory thresholding, yielding on par model accuracy. As such, we only store four MSBs per key vector element in transposable ReRAM arrays. The rest of LSBs can be stored on conventional ReRAM modules. Similarly, the elements of query and value vectors are stored on conventional ReRAM modules. Note that these modules do not need any support for in-memory computations and are solely used for storage[4].

To process the query vector $q_{1\times\mathcal{S}}$, the on-chip accelerator first transmits a subset of query vector MSBs to the transposable ReRAMs[5] that store the key matrix $\mathcal{K}^T_{d\times\mathcal{S}}$. A low-precision DAC converts the digital values of the query vector to

[4]We homogeneously use ReRAM for storage of queries and values and in-memory thresholding for simplicity. Another possibility can exploit a heterogeneous design, in which DRAM memories are used for query/value matrices and small ReRAM crossbar arrays for in-memory thresholding.

[5]The number of MSBs in query and key are identically set to 4-bits.

analog and feeds them into the ReRAM via wordlines. The transposable ReRAM array performs a low-precision vector-matrix multiplication in analog to calculate the Scores, which are produced after vertically applying an analog reduction sum per key vector. The next step performs in-memory thresholding using analog comparators. Note that the threshold values can either be set at the start of the computations or sent along with each q vector. Finally, after performing the analog comparisons, a voltage value (corresponding to a 1-bit digital value) flows through a series of 1-bit ADCs. The ADC outputs indicate the pruning state of their corresponding key vectors, where "1" means pruned.

The generated binary pruning vector is sent back to the on-chip accelerator, which subsequently gets translated into multiple memory requests to selectively fetch unpruned key and value vectors from their corresponding modules. Note that the pruning vectors for both key and value are completely identical. Upon receiving the first unpruned key vector, the accelerator can start recomputing Scores in full-precision. The same process repeats for the rest of the query vectors.

3.4 Sprint Memory Controller

Background. A memory controller receives a stream of memory access requests from core, generates their corresponding memory command stream. The memory controller consequently arbitrate the memory commands and schedule them to off-chip memory according to a scheduling policy. The technology of a memory (e.g. DRAM or ReRAM) dictates a set of timing constraints that must be satisfied by the memory controller between each issued memory command. To communicate data between core and off-chip memory, a sequence of memory commands generated by the memory controller are required. These commands collectively retrieve data from rows across multiple chips into their corresponding row buffers and select a column from the currently fetched retrieved data. A subsequent column access to the same row enjoys the row-buffer locality, hence, lowest access latency. However, the consecutive accesses between different rows are generally suffer from substantially higher access latency. The memory controller aims to schedule the

memory commands in order to maximize the row-buffer locality.

3.4.1 Data Layout Organization

We presume a similar organization as conventional memory subsystems for SPRINT. In general, optimizing the data layout organization for deep learning applications is straightforward because of their predictable memory access pattern. We observe the same pattern for SPRINT data layout organization. As explained, to support in-memory thresholding, we presume a non-interleaving data organization for $\mathcal{K}$s (similar to prior work [29, 16, 35]). That is, we store each vector of k (a column in $\mathcal{K}^T_{d\times\mathcal{S}}$) in one column of memory mat. Based on our observation (spatial locality between key elements, Section 3.1), we distribute the neighboring key vectors across different banks/channels. Our empirical results show that this distribution of key vectors provide a better utilization of memory bandwidth and reduce structural conflicts. Same data layout organization for value vectors Query matrix, on the other hand, does not need to follow this particular data layout organization. That is because each query vector is processed sequentially and after every query-key vector-matrix multiplication which provides sufficient time for the memory subsystem to handle the upcoming query read requests.

The final data layout organization requirement is for the MSB and LSB parts of key vectors. As described in Section 3.3.6, MSB and LSB parts of key vectors must be distributed across different type of ReRAM crossbar arrays, transposable and conventional respectively. This separation of MSB and LSB bits can be established statically before the computation starts. To effectively enable these special data layout organization, we can provide device-side allocation APIs so the user can specify different requirements for query/key/value matrices without exposing physical underlying structure of memory subsystem. Similar software support has been proposed in prior work [29, 14].

Scaling for large sequence length. One potential challenge to the proposed data layout organization and in-memory thresholding mechanism is posed by scalability. Specifically, as the embedding length of key vectors increases, applying the reduction sum across each column of ReRAM arrays may seem infeasible. This lim-

itation can be readily addressed by splitting the key vector into multiple adjacent ReRAM columns, similarly to [25]. With this circuit modification, the resulting analog current from the adjacent key vector splits can be subsequently merged and compared with the threshold value.

3.4.2 Memory Controller Microarchitecture

The on-chip memory controller designed for SPRINT is separated into a frontend and a backend engine. The frontend engine communicates with multiple cores, accepting memory requests, whereas the backend engine generates and issues commands to off-chip memory modules with respect to their timing constraints.

3.4.3 Memory Controller Execution Flow

Overview. The memory controller in SPRINT governs the tasks of in-memory thresholding and fetching the corresponding unpruned $d \times 1$ vectors of $\mathcal{K}^T_{d\times s}$ matrix. To complete these operations, the memory controller first sends a low-precision variant of q_i vector of size $1 \times d$ to $\mathcal{K}_{MSB}$ ReRAM banks. Each $\mathcal{K}_{MSB}$ ReRAM bank executes low-precision in-memory thresholding and generates a binary pruning vector of size s. The j^{th} element of the generated binary vector indicates whether to prune the j^{th} column of $\mathcal{K}^T_{d\times s}$ matrix (*i.e.*, '1' $\rightarrow$ pruned and '0' $\rightarrow$ unpruned). Upon receiving the binary pruning vector, the memory controller processes this vector and consequently issues a stream of read requests to fetch the unpruned vectors of $\mathcal{K}^T_{d\times s}$ matrix.

Spatial locality detection engine. To further reduce the data movement between off-chip memory and on-chip buffers, we design and integrate a spatial locality detection (SLD) engine in the front-end of the memory controller. The primary task of the engine is to detect and exploit spatial locality between the last and current binary pruning vectors associated with the attention score computations for adjacent query vectors (*i.e.*, $q^i_{1\times d}$ and $q^{i+1}_{1\times d}$). The advantages of are two folds: (1) "only" generating memory requests for key vectors that do not exists in on-chip key memory, hence reducing data transfer and memory contention, and (2)

bootstrapping the attention score ($\mathcal{Q} \times \mathcal{K}^T$) computations for the key vectors that already reside in on-chip key memory, hence minimizing the data transfer latency. The following equations describe the logic behind these two tasks given the last and current binary pruning vector:

$$\textbf{Task 1} \rightarrow \text{Memory Requests Vector} = \mathcal{P}^{t-1}_{1\times s} \wedge \overline{\mathcal{P}^{t}_{1\times s}} \tag{3.4}$$

$$\textbf{Task 2} \rightarrow \text{Spatial Locality Vector} = \overline{\mathcal{P}^{t-1}_{1\times s}} \wedge \overline{\mathcal{P}^{t}_{1\times s}} \tag{3.5}$$

where $\mathcal{P}^{t-1}_{1\times s}$ and $\mathcal{P}^{t}_{1\times s}$ represent binary pruning vectors associated with the last and current attention score computations at a given time point t, respectively.

Memory request generator engine. The main objective for the memory request generator (MRG) engine is to produce a potentially limited number of memory requests to fetch key vectors that do not currently reside in on-chip key memory. Each memory controller retains one MRG engine to produce the corresponding key vector addresses residing in that particular bank. At each cycle, a binary value is read from the memory request vector. If zero, it means that the corresponding key vector is not required for the current attention score computation; hence, bypassing memory request generation step. On the other hand, a one-value indicates that a key vector must be fetched from off-chip memory. Hence, a memory request with an address corresponding to the location of the desired key vector is generated.

To satisfy the key vector organization requirement, we decided to statically place the adjacent key vectors into memory modules attached to different channels. As such, to properly generate the key vector addresses, we equip each MRG with a base register and a shared up counter block. The base register indicates the starting key vector index located on a particular memory channel. The up counter starts from zero upon receiving a binary pruning vector and increases by the number of memory channels. We also equip each memory controller with a key index generator (KIG) engine, which has the exact same microarchitecture. However, in lieu of memory request vector, KIG engine operates on spatial locality vectors to generate the key vector addresses for SPRINT on-chip engines in order to bootstrap the attention score computations.

Memory commands and timing considerations. Supporting SPRINT style in-

memory thresholding into memory requires introducing additional memory commands and memory timing constraints. To enable in-memory thresholding in SPRINT, we introduce two additional memory commands, CopyQ and ReadP. CopyQ copies elements of query vector to in-memory query buffer, whereas ReadP reads elements of resulting binary pruning vector from in-memory pruning vector buffer. Depending on the bit-width of query and pruning vectors, the memory controller may issue one or more consecutive CopyQ and ReadP commands. Note that to initiate in-memory thresholding computations, we add one-bit in CopyQ command in which a one-value indicating the start of computations. Issuing other memory commands will be prohibited amid in-memory thresholding computations.

As you may observe, there is some similarities between CopyQ and ReadP commands and normal memory read and write, respectively, projecting a similar timing constraints as read/write commands. However, since CopyQ works with an isolated buffer from memory arrays, it neither requires tRP for row pre-charging, nor tRCD to activate a memory row. On the other hand, since consecutive CopyQ command still occupy data buses, we adhere to the tCL timing constraint. The scenario for ReadP is quite different as it communicates with the bank row buffers to read the resulting binary pruning vectors into on-chip buffers for further processing. Therefore, we conservatively follow the exact same timing constraints as memory read command for ReadP. For both introduced commands, burst CopyQ and ReadP follow the same timing constraints as normal burst memory read and write.

While the described scenarios for CopyQ and ReadP covers most of the required timing constraints, it still leaves one crucial timing constraints between adjacent CopyQ and ReadP commands. This timing, dubbed tAxTh, represents the number of cycles that each ReRAM crossbar requires to perform in-memory thresholding and producing the resulting pruning vector. Our circuit simulations show that this timing is < 8 cycles [8].

Power implications of in-memory thresholding. In addition to timing constraints, memory systems are also under power budget limitations. tFAW and tRRD represent the memory timing constraints linked to power budget. To ac-

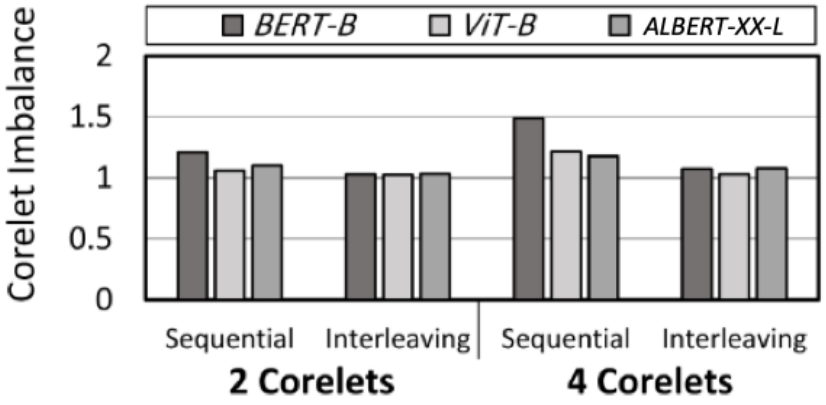


Figure 3.7: CORELET utilization imbalance with and without token interleaving across CORELETs.corelet in the figure should be capital

count for this power budget limitation, we model the analog in-memory thresholding circuit and estimate the power of analog comparators. Our simulation show that the overhead of additional analog circuitry for analog comparisons merely increase the total power budget by $< 0.4\%$ of total in-memory computation [36]. This power overhead has negligible implications on these timing constraints, hence we posit the nominal values for tFAW and tRRD in our simulations (similar to work [29, 28]).

3.5 On-Chip Accelerator

The SPRINT processor includes N CORELETs to enable a higher parallelism degree. Each CORELET consists of a QK-processing unit (QK-PU) and a V-processing unit (V-PU). QK-PU performs the $1 \times d$ dot product between $\mathcal{Q}$ and $\mathcal{K}$, whereas V-PU processes the $1 \times d$ dot product between the Softmax output and $\mathcal{V}$ in the digital domain. In addition, each CORELET has a small number of buffers to store unpruned key and value vectors. Note that the query vectors are processed in a stream manner, and thus do not need multi-entry buffers (Q-buf). Finally, each CORELET has its own look-up-tables to record which $\mathcal{K}$s and $\mathcal{V}$ are currently present on chip.

Workload balancing across CORELETs. SPRINT accelerator can simultaneously process multiple key vector sub-elements in each CORELET while the same query vector is distributed among all the CORELETs. As soon as the computations of one query and all of its associated keys complete, the computations of the next query can begin. In this design, the adjacent key vectors are assigned to different CORELETs, called *token-interleaving*. For example, given total four available CORELETs, SPRINT process $\mathcal{K}_{4n+i}$s in the i-th CORELET if the token is unpruned. This balances the workload across CORELETs while considering the spatial locality, by which the unpruned indices tend to appear in adjacent locations. Figure 3.7 shows the workload imbalance ratio. We calculate the imbalance ratio by dividing the maximum by the minimum numbers of assigned unpruned tokens per CORELET (i.e. the value of one implies ideal workload balance across CORELETs). We also analyzed alternative scenarios with two and four CORELETs. We observe that a system with four CORELETs suffers from poor balance as compared to a system with two CORELETs. This occurs because the unpruned key vectors can be more evenly distributed across a larger number of CORELETs, reducing utilization. The proposed workload distribution scheme considerably improves the utilization balance compared to the sequential token mapping, e.g. neighboring tokens belonging to the same CORELETs.

Handling data misses. To minimize the number of stalls due to data misses, the unpruned key vectors are proactively prefetched by the memory controller (as explained in Section 3.4.2). We also configure the main memory bandwidth (Table 3.1) to provide a new pair of $\mathcal{K}$ and $\mathcal{V}$ in burst mode to further reduce such stalls. Note that by leveraging the spatial locality between unpruned key vectors, on average, only 2.1% of the sequence length is required to be fetched between adjacent queries. This high data reuse drastically reduces the likelihood of data misses. When a rare data miss occurs, the computations for the next available key vector can proceed until the data miss is handled by the memory controller. We implement this bypassing of unavailable key vectors by adding a rotating pointer to key/value index buffers.

Sprint accelerator arithmetic operations. Once at least one key vector

resides in K-buf, the computation can start. At each cycle, SPRINT performs a dot-product between each subset of elements from key and query vectors. If all the key elements can not be processed during one compute iteration, SPRINT stores the partial sums in a register until the results are ready to be processed by a Softmax module. Similar to prior work [20, 37], we use a two look-up-tables method for exponent calculation. Afterwards, SPRINT stores the streaming outputs in FIFOs for accumulation. Once complete, each score is normalized to produce the corresponding probabilities. To balance the throughput between different stages of the pipeline, we employ two divider units. Finally, the computed score probabilities are used in V-PU to calculate the weighted sum of $\mathcal{V}$s. Note that the unpruned indices for key vectors can be used for the pruning of value vectors as well.

Two-dimensional sequence reduction. As introduced in 3.2.1, a large portion (e.g. 46% for the SQUAD dataset) of the total sequence length is futile due to zero-padding. Figure 3.2 illustrates the zero-padded (gray) area, which reduces the required output computation in both vertical and horizontal dimensions. Horizontally, the computation is reduced to $\mathcal{K}$ vectors per $\mathcal{Q}$, whereas vertically, it is reduced to $\mathcal{Q}$ vectors. We implement this mechanism by enabling the memory controller to filter out the read request for these masked regions.

Sprint accelerator design choice. The SPRINT accelerator does not employ a double-buffering scheme for on-chip memory in order to avoid the doubled cost of memory capacity. When the new data arrives from main memory, those are stored in a temporary small buffer. Meanwhile, a stall request is issued to initiate the write process into K-buf and V-buf. Note that, due to spatial locality across unpruned k elements for adjacent q vectors, the number of newly fetched $\mathcal{K}/\mathcal{V}$ is infrequent. Similar to prior work [37, 62, 20], SPRINT performs all the computations in 8-bit precision, except softmax that is carried out in 12-bits. For final attention score, we employ 16-bits precision.

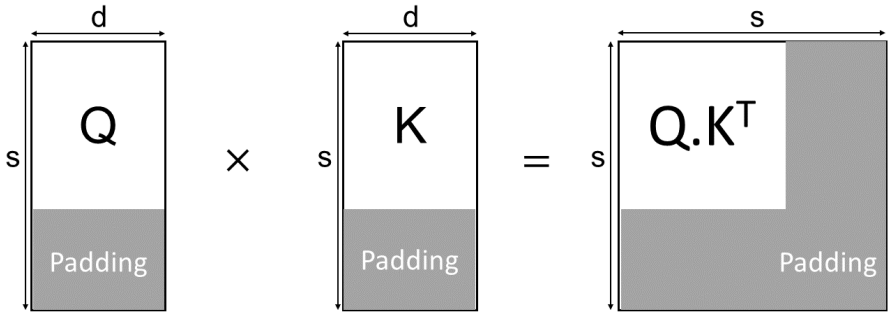


Figure 3.8: 2-dimensional masking to reduce the inconsequential computations.

3.6 Methodology and Evaluation

Benchmarks. The following models were used to evaluate the efficacy of SPRINT: BERT-Base (BERT-B) [27], BERT-Large (BERT-L) [27], ALBERT-X-Large (ALBERT-X-L) [34], ALBERT-XX-Large (ALBERT-XX-L) [34], ViT-Base (ViT-B) [10], and GPT-2-Large (GPT-2-L) [45]. We used the Stanford Question Answering Dataset (SQUAD) [46] to test BERT-B, BERT-L, ALBERT-X-L, ALBERT-XX-L, the WikiText-2[1] to test GPT-2-L and CIFAR10[31] dataset to test the ViT-Base. The sequence length (s) of 197, 384, 1280 is used for the CIFAR10, SQUAD and WikiText-2 dataset, respectively, whereas the same embedding size $d = 64$ is used for all the cases. On top of above datasets which exist today, we also create two more hypothetical synthetic models Synth1 and Synth2 with 2K and 4K sequences to estimate the projected benefit of SPRINT architecture for the potentially larger models.

Model fine-tuning for target benchmarks. As a baseline, We used the pretrained models from HuggingFace [65] and fine-tuned on each task using the hyperparameters reported in the original paper of each model [27, 34, 15]. The only hyperparameter we change is the batch size due to our limited GPU memory size, but the final fine-tuning result has no discernible change. These fine-tuned models are used as our baseline. By integrating the differentiable soft thresholding described in [37] into the HuggingFace transformer model [65], we find the optimal threshold for each attention layer during the task-specific fine-tuning process. As for the training hyperparamters, we use the same hyperparameter setting in BERT, ALBERT and ViT [27, 34, 15], except for the learning rate and the number of epochs.

Table 3.1: Hardware configurations of SPRINT.

Hardware modules	**Configurations for S-Sprint / M-Sprint / L-Sprint**
main memory BW	16×64-bit channels @ 1 GHz per CORELET
ReRAM array size	256×128 standard bitcell, 64×128 transposable array with 4-bit MLC
On-chip cache	16 / 32 / 64KB in total (= 8 / 16 / 32 banks), 128-bit port per bank
QK-PU / V-PU	1EA / 2EA / 4EA of 1-D 64 (=$\mathcal{D}$) way 8×8-bit MAC array
Softmax	1EA / 2EA / 4EA of 12-bit input, 8-bit output, 2EA of LUTs: 125 MB each

The search space for the threshold learning rate is $\{2e^{-3}, 2e^{-4}, 2e^{-5}\}$, whereas the search space of $\{2e^{-5}, 2e^{-6}\}$ used for all other parameters. The number of epochs ranges from 1 to 3 for different tasks. We run all the experiments mentioned above with PyTorch v1.10 [44] on an Nvidia RTX 3090 GPU.

This results in the pruning rate of 74.6%, 75.5%, 65.1%, 73.1%, 64.4%, and 73.9% for BERT-B, BERT-L, ALBERT-XL, ALBERT-XXL with SQUAD dataset, ViT-B with CIFAR10 dataset, and GPT-2-L with WikiText-2. For the Synth1 and Synth2, 75% pruning rate and 50% padding ratio are assumed for the following simulations. The estimated main memory access when switching to new query vector for Synth1 and Synth2 are obtained by scaling up the numbers from BERT-B based on the sequence length difference.

Sprint hardware simulations. Table 3.1 lists the design parameters of SPRINT for three studied configurations: (1) S-SPRINT: a CORELET with 16KB total on-chip memory capacity, (2) M-SPRINT: two CORELETs with 32KB, and (3) L-SPRINT: four CORELETs with 64KB to analyze the impact over different parallelism and on-chip memory size. We tested above configurations by using Cadence Genus 19.1 [6] for the logic synthesis and Cadence Innovus 19.1 [7] for the placement/routing (PnR) of digital blocks with 65 nm TSMC general-purpose standard cell library. These digital blocks are generated to meet the target frequency of 1GHz from the post-layout simulations. The SRAM on-chip memory was generated by ARM Memory Compiler with High density 65 nm single-port SRAM (version r0p0) [2]. On the other hand, ReRAM crossbar in-memory operation consumes 0.10 pJ / MAC in 65 nm including the digtal-to-analog conversion [8] whereas ReRAM standard read and write operation spend 3.1 pJ / bit and 24.4 pJ / bit, respectively [18]. Each analog comparator consumes 41 fJ [36]. An extensive study of ReRAM in-memory computing from HP Lab [23] has shown that

64-tap in-memory dot-product delivers 5-bit equivalent output accuracy in terms of signal-to-noise ratio after including all the error sources. To emulate the limited accuracy of the in-memory thresholding, same error models in Section 3.3.5 is used with $b = 5$. Based on above component analysis, we develop an SPRINT simulator to provide the total delay cycles and number of accesses to the memory for both SPRINT and baseline accelerators in consideration of the limited on-chip memory capacity, main memory bandwidth, and pruning behavior with a special locality from each workload.

Baseline architecture. As a baseline, we employed the same configuration of S-SPRINT, M-SPRINT, and L-SPRINT, but without the in-memory pruning, proposed memory controller, and two-dimensional computing reduction for the padded sequences. We compare SPRINT and the baseline at iso-setups including the same frequency, the number of processing elements, on-chip memory capacity, and bit widths for all the input and output of digital logic blocks.

Comparison to A^3, SpAtten, and LeOPArd. A^3 [20], SpAtten [62], and LeOPArd [37] also support the run-timing pruning to minimize the required computation. A^3 thresholds after processing a limited number of k vectors from the sorted queue in a magnitude order to minimize the run-timing pruning overhead. But, A^3 does not consider the data movement cost from the main memory assuming enough on-chip memory capacity. LeOPArd performs the gradient-based training to co-optimize the model accuracy and pruning rate by tuning the pruning threshold automatically during the training instead of empirical methods. Again, LeOPArd does not consider the cost from main memory access. SpAtten proposed a cascaded pruning to exclude the redundant heads and tokens for all the subsequent layers once those are pruned in the previous layer. SpAtten reduces the DRAM access cost for GPT-2, but not for other models assuming enough on-chip capacity. Due to the lack of available raw performance/energy results for individual workloads and simulation infrastructures of the accelerators, commensurate with comparison methodology of SpAtten [62], we use throughput (GOPs / s), energy efficiency (GOPs / J), and area efficiency (GOPs / s / mm^2) to provide the best effort analyses.

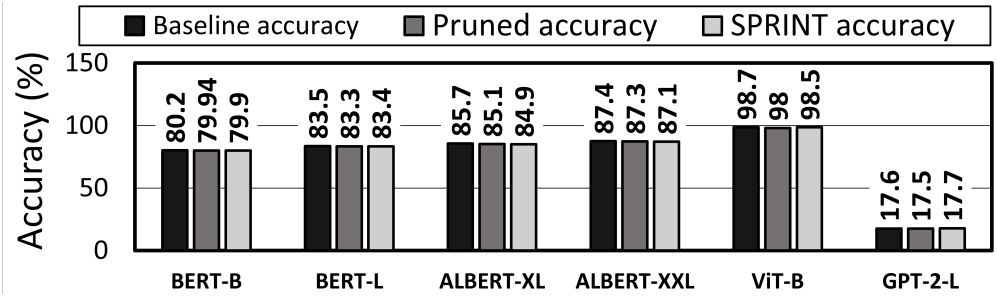


Figure 3.9: Accuracy from software vs. LeOPArd with analog in-memory run-time pruning. Here, GPT-2-L accuracy is measured as a perplexity metric.

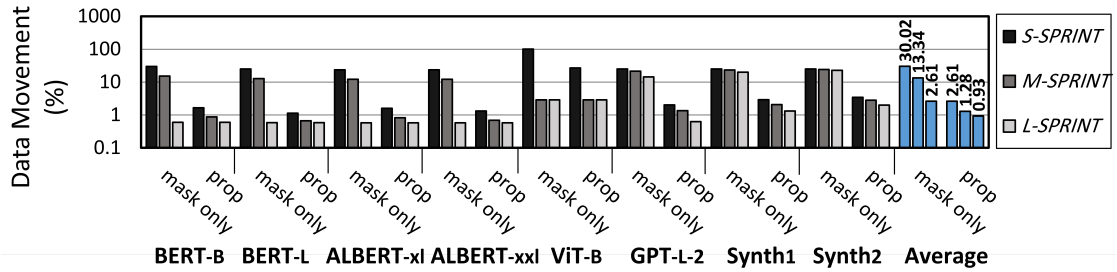


Figure 3.10: Total data movement from main memory normalized to that of S-Baseline configuration.

3.6.1 Accuracy and Performance

Impacts on model accuracy from in-memory pruning. The accuracy of the tested workload from Sprint architecture is described in Figure 3.9. To quantify the impact from the run-timing pruning and the analog in-memory thresholding's inaccuracy separately, we measured three cases: 1) the baseline accuracy [65], 2) with the only run-timing pruning, and 3) run-timing pruning including the limited accuracy from the in-memory thresholding (equivalent to 5-bit at the output). The accuracy degradation from the run-time pruning is 0.3% on average with maximum of 0.8% compared to the baseline accuracy whereas the Sprint has the degradation of 0.3% on average with the maximum of 0.8% accuracy compared to the baseline. We can conclude that both run-time pruning and the inaccuracy from the approximate thresholding have minimal impact on the model accuracy.

Main memory data movement analysis. Figure 3.10 shows the total amount of data movement from the main memory to the processor during pro-

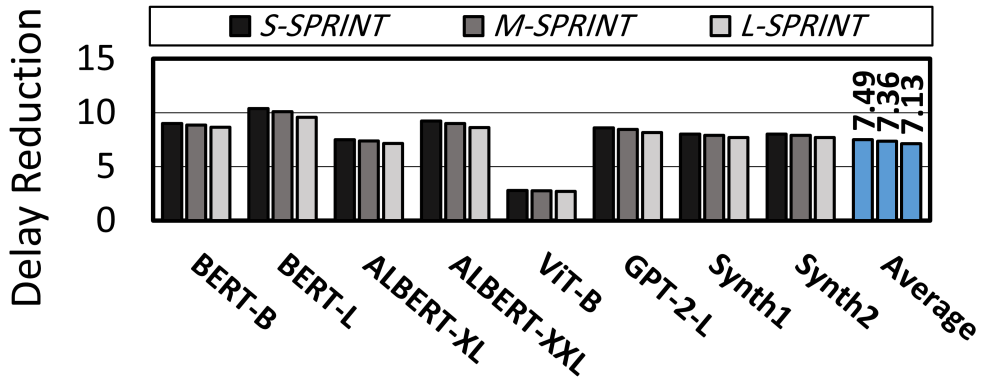


Figure 3.11: Speed-up comparison to Baseline design.

cessing a single head in two configurations: 1) with sequence reduction for the padded area, 2) with run-timing pruning on top of the sequence reduction of the padded area. All the numbers are normalized to that of S-Baseline in each testbench. In all the 48 studied configurations, the proposed scheme provides a significant reduction in the data movement requiring, on average, 2.6%, 1.3%, and 0.9% in S-SPRINT, M-SPRINT and L-SPRINT, respectively, compared to the S-Baseline. The benefit varies across workloads due to their different pruning rate and the portion of padded area, e.g., BERT-B has higher data movement reduction due to its 46% padded area and 74.6% pruning rate compared to ViT with 64.3% pruning rate with zero padding. The sequence reduction only scenario has a limited benefit still requiring a data movement of 30.0%, 13.3%, and 2.6% in S-SPRINT, M-SPRINT, and L-SPRINT, respectively, on average, as compared to the S-Baseline case. The only exception is observed in ViT-B due to the lack of zero padding. In all the configurations, L-SPRINT requires fewer main memory access than S-SPRINT and M-SPRINT due to the higher on-chip memory capacity. Again, ViT-B shows the exception because M-SPRINT has already sufficient memory capacity to store entire (197) sequence length. The gap among S-, M-, L-SPRINT configurations is smallest in Synth1 and Synth2, where sequence length is very long. Thus, even L-SPRINT model can accommodate only highly limited fraction of the entire sequence length, e.g. 12.5% in Synth2. For the same reason, the data movement reduction is less significant in Synth models compared to others as those cannot contain enough number of correlated tokens in their tight memory space.

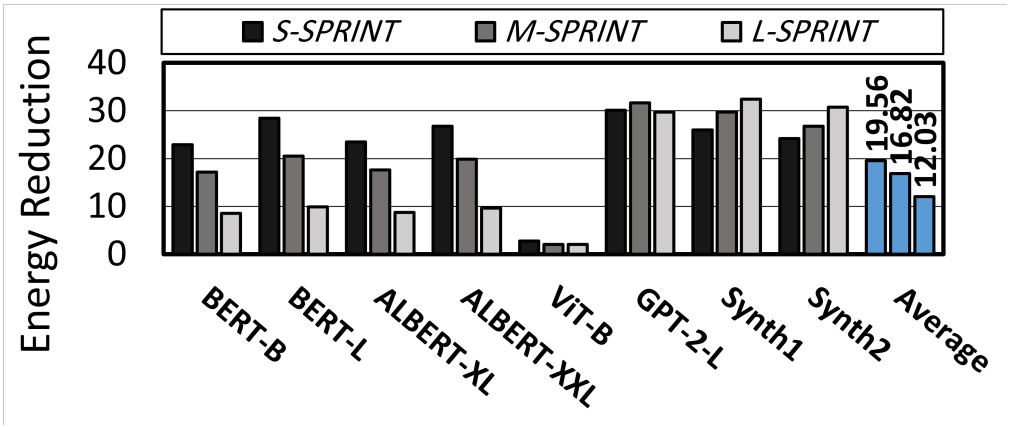


Figure 3.12: Total energy reduction compared to Baseline.

Performance and energy comparison. We measure the speed-up achieved by SPRINT in Figure 3.11 compared to the Baseline design across all the 24 studied task. S-, M-, and L-SPRINT achieve 7.5× , 7.4× , and 7.1× speed-up, respectively, on average over the Baseline design by skipping the majority of potential computing cycles for the redundant tokens via the in-memory run-time pruning. The speed-up benefit diminishes in L-SPRINT slightly (<8% compared to S-SPRINT) because the workload utilization is not perfectly balanced across CORELETs as shown in Figure 3.7 even after the proposed k vector distribution. BERT-L enjoys the maximum benefits with 9.6 - 10.4× speed-up while ViT-B has minimum improvement of 2.7 - 2.8×. This is because of the different pruning rates and portion of padded area in those models as mentioned above.

Figure 3.12 shows the energy reductions achieved by SPRINT, including on-chip accelerator and ReRAM-based main memory, compared to the Baseline for the three configurations. The energy reductions of 19.6× for S-SPRINT, 16.8× for M-SPRINT, and 12.0× for L-SPRINT are achieved. The S-SPRINT achieves the largest improvement because the proportion of main memory access out of the total energy is larger than the other cases due to the highly constrained memory capacity and frequent memory access. This leads to more improvement by the proposed technique which reduces the data movement effectively. On the other hand, Synth1 and Synth2 show the exception in this trend because even L-SPRINT can contain only very few fraction of the entire sequence, e.g. 12.5% as mentioned above whereas 100% in BERT-B in L-SPRINT. In such a regime, where the memory

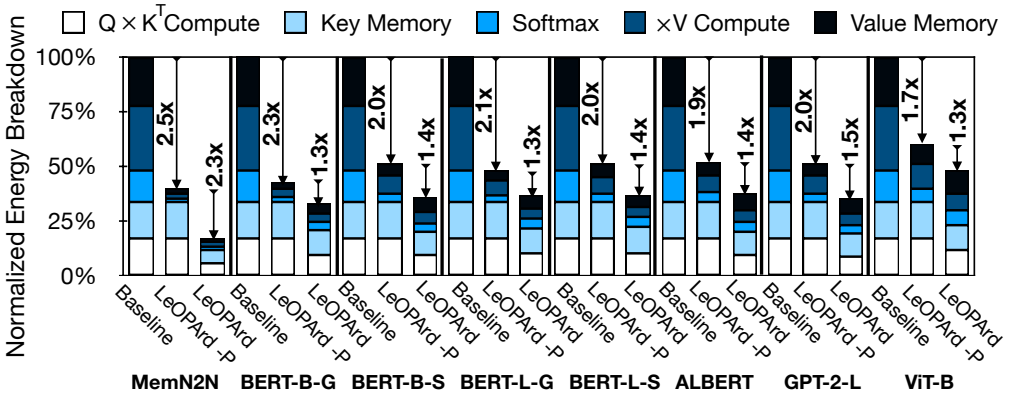


Figure 3.13: M-SPRINT's energy breakdown normalized to baseline. First bar and second bar represent baseline and M-SPRINT, respectively.

capacity is significantly limited, the larger memory provides the more room to fetch the correlated data together increasing the chance of data re-use. For this reason, L-SPRINT achieves more energy benefit compared to S- and M-SPRINT in Synth models. The energy benefit is greater in Synth1 and Synth2 models than the other cases as those require more frequent data access due to their large sequence length so that the benefit by SPRINT is magnified. In contrast, ViT shows the minimum benefit due to its small sequence length, and thus infrequent data access.

Energy consumption breakdown. Figure 3.13 provides the detailed energy breakdown of M-SPRINT including following parts: 1) main memory read, 2) main memory write, 3) in-memory pruning, 4) on-chip memory (K-buf and V-buf) read, 5) on-chip memory write, and computations in 6) QK-PU, 7) V-PU, and 8) Softmax. Roughly 50% of the energy consumption of the Baseline design is from ReRAM read operations in all the benchmarks other than ViT-B, whose sequence length is tiny. It is shown SPRINT reduces the main memory access energy effectively in all the workloads.SPRINT's in-memory pruning allows to skip the large portion of computations as well, which results in remarkable reduction in computation energy by QK-PU, V-PU, and Softmax. The SPRINT architecture also reduces the writing energy of main memory to some degree by avoiding the ReRAM writing for the zero padded area. Despite this reduction, main memory writing takes the largest energy portion out of all the components in SPRINT (except ViT-B) due to more aggressive reductions in the other energy components. The overhead of in-memory pruning including peripheral circuitry is negligible taking only 4% out

Table 3.2: SPRINT performance comparison with prior arts.

Metric (unit)	A^3	SpAtten	LeOPArd	M-Sprint (on-chip)	M-Sprint
Sequence Length	50 - 384	384 - 1024	50 - 4096	128 - 4096	128 - 4096
Process (nm)	40	40	65	65	65
Area (mm^2)	2.1	1.6	3.5	1.8	1.8
Key Buffer (KB)	20	24	48	16	16
Value Buffer (KB)	20	24	64	16	16
GOPs / s	518.0	360.0	574.1	518.0	1816.2
GOPs / J	4709.1	382.0	519.3	524.9	257.5
GOPs / s / mm^2	249.0	238.0	165.5	290.0	1003.6
Mem Cost Included	**NO**	**YES**	**NO**	**NO**	**YES**

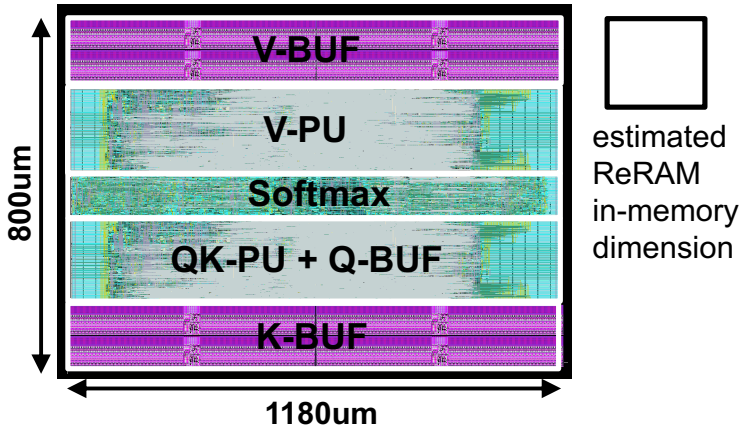


Figure 3.14: S-SPRINT on-chip accelerator layout with estimated ReRAM in-memory area overhead [59].

of the entire energy due to its highly parallel and low-voltage analog operations. This implies that the benefits from the in-memory pruning far outweighs its own overhead.

Comparison with A^3, SpAtten, and LeOPArd. Table 3.3 lists the details of prior works and M-SPRINT architecture in terms of throughput (GOPs / s), energy efficiency (GOPs / J), and area efficiency (GOPs / s / mm^2). For fair comparison, we also included the area from the in-memory thresholding [59], which takes only 3% out of total M-SPRINT area. Due to the absence of reported results in A^3, we calculated above results in Table 3.3 given the frequency and power numbers obtained from [20]. The prior arts considered the scenario of enough on-chip memory with minimal consideration of the dram access cost. On the other hand, SPRINT

Table 3.3: SPRINT performance comparison with prior arts.

Metric (unit)	A^3	SpAtten	LeOPArd	M-Sprint
Sequence Length	50 - 384	384 - 1024	50 - 1024	128 - 4096
Process (nm)	40	40	65	65
Area (mm^2)	2.1	1.6	3.5	1.9
Key Buffer (KB)	20	24	48	16
Value Buffer (KB)	20	24	64	16
GOPs / s	518.0	360.0	574.1	1816.2
GOPs / J	4709.1	382.0	519.3	257.5
GOPs / s / mm^2	249.0	238.0	165.5	973.5
Mem Cost Included	**NO**	**YES**	**NO**	**YES**

includes all the costs from the frequent main memory access assuming the limited on-chip memory scenario by considering ¿ 4× longer sequences than prior arts.

Compared to any prior works, M-SPRINT achieves the best GOPs / s and GOPs / s / mm^2 even including the main memory access cost due to it's in-memory pruning. Compared to A^3, M-SPRINT achieves 3.5× and 3.9× improvements in GOPs / s and GOPs / s / mm^2 respectively. However, it achieves 18× lower GOPs / J. This is because the DRAM access read and write costs are not considered in the results of A^3 in addition to the lower process technology (40 nm) in A^3. Similarly, M-SPRINT achieves 3.2× higher GOPs / s and 5.9× higher GOPs / s / mm^2 than LeOPArd, while 2.0× lower GOPs / J is observed in M-SPRINT due to the consideration of main memory access cost. Although SpAtten incorporated the DRAM access cost, M-SPRINT employs insufficient on-chip memory capacity requiring more frequent main memory access. Despite the fact, M-SPRINT still delivers 5.0× and 4.1× enhancements in GOPs / s and GOPs / s / mm^2, respectively, while achieving 1.5× worse GOPs / J. This difference in GOPs / J is negligible in consideration of the different process technology, e.g. 40 nm in SpAtten vs. 65 nm in SPRINT. The benefits that are gained from GOPs / s and GOPs / s / mm^2 are based on the early stage in-memory pruning by leveraging the spatial locality. We expect greater benefit compared to the prior arts when the workloads with very long sequence are tested equivalently.

SPRINT on-chip accelerator and ReRAM in-memory Area. Figure 3.14 shows the S-SPRINT layout in a 65 nm process which occupies 1.8×0.8 mm^2 including total 16KB SRAM. The layout estimation of ReRAM in-memory, including 64×128 transposable array and other peripheral circuitry is also shown in Figure 3.14. Due

to the inherent high-density of ReRAM, the area overhead takes only around 6% in the S-SPRINT case. This portion can be further reduced in M- or L-SPRINT by amortizing the overhead across cores.

Chapter 3 is adapted from the material that has been submitted for publication as it appears in Amir Yazdanbakhsh, Ashkan Moradifirouzabadi, Zheng Li, Mingu Kang. "Sparse Attention Acceleration with Approximate In-Memory Pruning." The thesis author was the primary investigator and author of this paper.

Chapter 4

Conclusion and Future Work

Transformers through the self-attention mechanism have triggered an exciting new wave in machine learning, notably in Natural Language Processing (NLP). The self-attention mechanism computes pairwise correlations among all the words in a subtext. This task is both compute and memory intensive and has become one of the key challenges in realizing the full potential of attention models. However, deploying these models on hardware with limited-resource can be challenging. To address this, two accelerators, LeOPArd and Sprint are introduced in Chapter 2 and Chapter 3 in this thesis.

LeOPArd focuses on pruning the inconsequential scores at runtime through a thresholding mechanism. The work in Chapter 2 exclusively formulated the pruning threshold finding as a gradient-based optimization problem. This formulation strikes a formal and analytical balance between model accuracy and computation reduction. To maximize the performance gains from thresholding, a bit-serial architecture is shown in Chapter 2 to enable an early-termination atop pruning with no repercussions to model accuracy. These techniques synergistically yield significant benefits both in terms of speedup and energy savings across various transformer-based models on a range of NLP and vision tasks. The application of the proposed mathematical formulation of identifying threshold values and its cohesive integration into the training loss is broad and can potentially be adopted across a wide range of compute reduction techniques.

Sprint focuses on solving the on-chip memory limitation and data-movement

overhead of self-attention Computation. It harnesses the inherent parallelism of ReRAM crossbar arrays to compute the attention scores in low-precision format. The resulting attention scores cross a lightweight analog thresholding circuitry, which dynamically prune the inconsequential ones, based on the techniques in Chapter 2. To mitigate the negative repercussion of approximate ReRAM computations on model accuracy, SPRINT recomputes the sparse attention scores for the few fetched data in digital. The combined in-memory pruning and on-chip recomputation of the relevant attention scores reduce the quadratic complexity of self-attention mechanism into a merely linear one.

In the future, it is possible to leverage the retraining strategy and the regularization term introduced in Chapter 2 to control the pruning rate to be the same on different tasks for the same model. Then, more hardware designs can be developed to exploit the uniform pruning rate of the algorithm.

Bibliography

[1] The WikiText Long Term Dependency Language Modeling Dataset. https://blog.salesforceairesearch.com/the-wikitext-long-term-dependency-language-modeling-dataset/, 2021. Accessed: 2021-11-08.

[2] ARM. Artisan Memory Compilers. https://developer.arm.com/ip-products/physical-ip/embedded-memory, 2021. Accessed: 2021-11-08.

[3] Azarian, K., Bhalgat, Y., Lee, J., and Blankevoort, T. Learned Threshold Pruning.

[4] Aziza, H., Hamdioui, S., Fieback, M., Taouil, M., Moreau, M., Girard, P., Virazel, A., and Coulié, K. Multi-Level Control of Resistive RAM (RRAM) Using a Write Termination to Achieve 4 Bits/Cell in High Resistance State. *Electronics* (2021).

[5] Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. 3rd International Conference on Learning Representations, ICLR 2015 ; Conference date: 07-05-2015 Through 09-05-2015.

[6] Cadence. Genus Synthesis Solution. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html, 2021. Accessed: 2021-11-08.

[7] Cadence. Innovus Implementation System. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html, 2021. Accessed: 2021-11-08.

[8] Cai, F., Correll, J. M., Lee, S. H., Lim, Y., Bothra, V., Zhang, Z., Flynn, M. P., and Lu, W. A fully integrated reprogrammable memristor–cmos system for efficient multiply–accumulate operations. *Nature Electronics* (2019), 290–299.

[9] Chang, M.-F., Chiu, P.-F., and Sheu, S.-S. Circuit design challenges in embedded memory and resistive RAM (RRAM) for mobile SoC and 3D-IC.

In *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)* (2011).

[10] Chen, T., Cheng, Y., Gan, Z., Yuan, L., Zhang, L., and Wang, Z. Chasing Sparsity in Vision Transformers: An End-to-End Exploration.

[11] Chen, T., Cheng, Y., Gan, Z., Yuan, L., Zhang, L., and Wang, Z. Chasing sparsity in vision transformers:an end-to-end exploration, 2021.

[12] Chi, P., Li, S., Xu, C., Zhang, T., Zhao, J., Liu, Y., Wang, Y., and Xie, Y. Prime: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-based Main Memory. In *ISCA* (2016).

[13] Chi, P., Li, S., Xu, C., Zhang, T., Zhao, J., Liu, Y., Wang, Y., and Xie, Y. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ACM SIGARCH Computer Architecture News* (2016), vol. 44, IEEE Press, pp. 27–39.

[14] Choi, C. Multi-Stream Write SSD. *Flash Memory Summit* (2016).

[15] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *ICLR* (2021).

[16] Farmahini-Farahani, A., Ahn, J. H., Morrow, K., and Kim, N. S. NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules. In *HPCA* (2015), pp. 283–295.

[17] Gale, T., Elsen, E., and Hooker, S. The State of Sparsity in Deep Neural Networks.

[18] Grossi, A., Vianello, E., Sabry, M. M., Barlas, M., Grenouillet, L., Coignus, J., Beigne, E., Wu, T., Le, B. Q., Wootters, M. K., Zambelli, C., Nowak, E., and Mitra, S. Resistive ram endurance: Array-level characterization and correction techniques targeting deep learning applications. *IEEE Transactions on Electron Devices 66*, 3 (2019), 1281–1288.

[19] Halawani, Y., Mohammad, B., Lebdeh, M. A., Al-Qutayri, M., and Al-Sarawi, S. F. ReRAM-based In-memory Computing for Search Engine and Neural Network Applications. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2019).

[20] Ham, T. J., Jung, S. J., Kim, S., Oh, Y. H., Park, Y., Song, Y., Park, J.-H., Lee, S., Park, K., Lee, J. W., et al. Aˆ3: Accelerating Attention Mechanisms in Neural Networks with Approximation. In *HPCA* (2020).

[21] Ham, T. J., Lee, Y., Seo, S. H., Kim, S., Choi, H., Jung, S. J., and Lee, J. W. ELSA: Hardware-Software Co-design for Efficient, Lightweight Self-Attention Mechanism in Neural Networks. In *ISCA* (2021).

[22] He, Z., Lin, J., Ewetz, R., Yuan, J.-S., and Fan, D. Noise Injection Adaption: End-to-end ReRAM Crossbar Mon-Ideal Effect Adaption for Meural Network Mapping. In *DAC* (2019).

[23] Hu, M., Strachan, J. P., Li, Z., Grafals, E. M., Davila, N., Graves, C., Lam, S., Ge, N., Yang, J. J., and Williams, R. S. Dot-product Engine for Neuromorphic Computing: Programming 1T1M Crossbar to Accelerate Matrix-Vector Multiplication. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)* (2016).

[24] Joardar, B. K., Doppa, J. R., Pande, P. P., Li, H., and Chakrabarty, K. AccuReD: High Accuracy Training of CNNs on ReRAM/GPU Heterogeneous 3-D Architecture. *IEEE TCAD* (2021).

[25] Kang, M., Gonugondla, S. K., Patil, A., and Shanbhag, N. R. A Multi-functional In-memory Inference Processor using a Standard 6T SRAM Array. *IEEE Journal of Solid-State Circuits* (2018).

[26] Katharopoulos, A., Vyas, A., Pappas, N., and Fleuret, F. Transformers are RNNs: Fast AutoRegressive Transformers with Linear Attention. In *ICML* (2020).

[27] Kenton, J. D. M.-W. C., and Toutanova, L. K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT* (2019).

[28] Kim, B., Chung, J., Lee, E., Jung, W., Lee, S., Choi, J., Park, J., Wi, M., Lee, S., and Ahn, J. H. MViD: Sparse Matrix-Vector Multiplication in Mobile DRAM for Accelerating Recurrent Neural Networks. *IEEE Transactions on Computers* (2020).

[29] Kim, H., Park, H., Kim, T., Cho, K., Lee, E., Ryu, S., Lee, H.-J., Choi, K., and Lee, J. GradPIM: A Practical Processing-in-DRAM Architecture for Gradient Descent. In *HPCA* (2021).

[30] Kim, S., Shen, S., Thorsley, D., Gholami, A., Hassoun, J., and Keutzer, K. Learned token pruning for transformers. *arXiv preprint arXiv:2107.00910* (2021).

[31] Krizhevsky, A., and Hinton, G. Learning Multiple Layers of Features from Tiny Images. *Computer Science Department, University of Toronto, Tech. Rep* (2009).

[32] Krogh, A., and Hertz, J. A. A Simple Weight Decay can Improve Generalization. In *NIPS* (1992).

[33] Kullback, S., and Leibler, R. A. On Information and Sufficiency. *The annals of mathematical statistics* (1951).

[34] Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *ICLR* (2019).

[35] Lee, J., Ahn, J. H., and Choi, K. Buffered Compares: Excavating the Hidden Parallelism Inside DRAM Architectures with Lightweight Logic. In *DATE* (2016).

[36] Li, W., Xu, P., Zhao, Y., Li, H., Xie, Y., and Lin, Y. TIMELY: Pushing Data Movements and Interfaces in PIM Accelerators towards Local and in Time Domain. In *ISCA* (2020).

[37] Li, Z., Ghodrati, S., Yazdanbakhsh, A., Esmaeilzadeh, H., and Kang, M. Accelerating Attention through Gradient-Based Learned Runtime Pruning. In *ISCA* (2022).

[38] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. RoBERTa: A Robustly Optimized BERT Pretraining Approach.

[39] Lou, Q., Wen, W., and Jiang, L. 3DICT: A Reliable and QoS Capable Mobile Process-in-Memory Architecture for Lookup-Based CNNs in 3D XPoint ReRAMs. In *ICCAD* (2018).

[40] Louizos, C., Welling, M., and Kingma, D. P. Learning Sparse Neural Networks through L_0 Regularization.

[41] Mittal, S. A Survey of ReRAM-based Architectures for Processing-In-Memory and Neural Networks. *Machine learning and knowledge extraction* (2018).

[42] Murphy, K. P. *Machine Learning: A Probabilistic Perspective*. MIT press, 2012.

[43] Niu, D., Xu, C., Muralimanohar, N., Jouppi, N. P., and Xie, Y. Design Trade-Offs for High Density Cross-Point Resistive Memory. In *ISLPED* (2012).

[44] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison,

A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 2019.

[45] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language Models are Unsupervised Multitask Learners. *OpenAI blog* (2019).

[46] Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. SQUAD: 100,000+ Questions for Machine Comprehension of Text.

[47] Robbins, H., and Monro, S. A Stochastic Approximation Method. *The annals of mathematical statistics* (1951).

[48] Shafiee, A., Nag, A., Muralimanohar, N., Balasubramonian, R., Strachan, J. P., Hu, M., Williams, R. S., and Srikumar, V. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (2016), pp. 14–26.

[49] Song, L., Qian, X., Li, H., and Chen, Y. Pipelayer: A pipelined reram-based accelerator for deep learning. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on* (2017), IEEE, pp. 541–552.

[50] Song, L., Zhuo, Y., Qian, X., Li, H., and Chen, Y. GraphR: Accelerating Graph Processing using ReRAM. In *HPCA* (2018).

[51] Srinivas, S., Subramanya, A., and Venkatesh Babu, R. Training Sparse Neural Networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition Workshops* (2017).

[52] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The journal of machine learning research* (2014).

[53] Stillmaker, A., and Baas, B. Scaling Equations for the Accurate Prediction of CMOS Device Performance from 180 nm to 7 nm. *Integration* (2017).

[54] Stojcevski, A., Le, H. P., Singh, J., and Zayegh, A. Flash adc architecture. *Electronics letters 39*, 6 (2003), 501–502.

[55] Sukhbaatar, S., szlam, a., Weston, J., and Fergus, R. End-To-End Memory Networks. In *NIPS* (2015).

[56] Tambe, T., Hooper, C., Pentecost, L., Jia, T., Yang, E.-Y., Donato, M., Sanh, V., Whatmough, P., Rush, A. M., Brooks, D., et al. EdgeBERT: Sentence-level Energy Optimizations for Latency-Aware Multi-Task NLP Inference. In *MICRO* (2021).

[57] Transformers: Opening New Age of Artifical Intelligence Ahead. Adc performance survey 1997-2016. https://www.analyticsinsight.net/transformers-opening-new-age-of-artificial-intelligence-ahead/, 2021. Accessed: 2022-04-21.

[58] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is All You Need. In *NeurIPS* (2017).

[59] Wan, W., Kubendran, R., Eryilmaz, S. B., Zhang, W., Liao, Y., Wu, D., Deiss, S., Gao, B., Raina, P., Joshi, S., et al. 33.1 a 74 tmacs/w cmos-rram neurosynaptic core with dynamically reconfigurable dataflow and in-situ transposable weights for probabilistic graphical models. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)* (2020), IEEE, pp. 498–500.

[60] Wan, W., Kubendran, R., Eryilmaz, S. B., Zhang, W., Liao, Y., Wu, D., Deiss, S., Gao, B., Raina, P., Joshi, S., et al. A 74 TMACS/W CMOS-RRAM Neurosynaptic Core with Dynamically Reconfigurable Dataflow and In-situ Transposable Weights for Probabilistic Graphical Models. In *ISSCC* (2020).

[61] Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding.

[62] Wang, H., Zhang, Z., and Han, S. SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning. In *HPCA* (2021).

[63] Wang, Y., Zhu, Z., Chen, F., Ma, M., Dai, G., Wang, Y., Li, H., and Chen, Y. Rerec: In-ReRAM Acceleration with Access-Aware Mapping for Personalized Recommendation. In *ICCAD* (2021).

[64] Weston, J., Bordes, A., Chopra, S., Rush, A. M., van Merriënboer, B., Joulin, A., and Mikolov, T. Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks. *arXiv preprint arXiv:1502.05698* (2015).

[65] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. HuggingFace's Transformers: State-of-the-Art Natural Language Processing.

[66] Wu, M.-C., Jang, W.-Y., Lin, C.-H., and Tseng, T.-Y. A Study on Low-Power, Nanosecond Operation and Multilevel Bipolar Resistance Switching in Ti/ZrO2/Pt Nonvolatile Memory with 1T1R Architecture. *Semiconductor Science and Technology - SEMICOND SCI TECHNOL* (2012).

[67] Yang, T., Li, D., Han, Y., Zhao, Y., Liu, F., Liang, X., He, Z., and Jiang, L. Pimgcn: A reram-based pim design for graph convolutional network acceleration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*.

[68] Yao, P., Wu, H., Gao, B., Eryilmaz, S. B., Huang, X., Zhang, W., Zhang, Q., Deng, N., Shi, L., Wong, H.-S. P., et al. Face classification using electronic synapses. *Nature communications 8* (2017), 15199.

[69] Ye, Z., Guo, Q., Gan, Q., Qiu, X., and Zhang, Z. BP-Transformer: Modelling Long-Range Context via Binary Partitioning.

[70] Yu, S., Wu, Y., and Wong, H.-S. P. Investigating the Switching Dynamics and Multilevel Capability of Bipolar Metal Oxide Resistive Switching Memory. *Applied Physics Letters* (2011).

[71] Yuan, G., Behnam, P., Li, Z., Shafiee, A., Lin, S., Ma, X., Liu, H., Qian, X., Bojnordi, M. N., Wang, Y., and Ding, C. FORMS: Fine-grained Polarized ReRAM-based In-situ Computation for Mixed-signal DNN Accelerator. In *ISCA* (2021).

[72] Zagoruyko, S., and Komodakis, N. Wide Residual Networks.

[73] Zhang, L., Strukov, D., Saadeldeen, H., Fan, D., Zhang, M., and Franklin, D. SpongeDirectory: Flexible Sparse Directories Utilizing Multi-Level Memristors. In *PACT* (2014).

[74] Zou, H., and Hastie, T. Regularization and Variable Selection via the Elastic Net. *Journal of the royal statistical society: series B (statistical methodology)* (2005).