Mokey: Enabling Narrow Fixed-Point Inference for Out-of-the-Box Floating-Point Transformer Models

Ali Hadi Zadeh
University of Toronto, Vector Institute
Toronto, Canada
hadizade@ece.utoronto.ca

Mostafa Mahmoud
University of Toronto
Toronto, Canada
mostafa.mahmoud@mail.utoronto.ca

Ameer Abdelhadi
University of Toronto
Toronto, Canada
ameer.abdelhadi@utoronto.ca

Andreas Moshovos
University of Toronto, Vector Institute
Toronto, Canada
moshovos@ece.utoronto.ca

***Abstract*—Increasingly larger and better Transformer models keep advancing state-of-the-art accuracy and capability for Natural Language Processing applications. These models demand more computational power, storage, and energy. *Mokey* reduces the footprint of state-of-the-art 32-bit or 16-bit floating-point transformer models by quantizing all values to 4-bit indexes into dictionaries of representative 16-bit fixed-point centroids. *Mokey* does not need fine-tuning, an essential feature as often the training resources or datasets are not available to many. Exploiting the range of values that naturally occur in transformer models, *Mokey* selects centroid values to also fit an exponential curve. This unique feature enables *Mokey* to replace the bulk of the original multiply-accumulate operations with narrow 3b fixed-point additions resulting in an area- and energy-efficient hardware accelerator design. Over a set of state-of-the-art transformer models, the *Mokey* accelerator delivers an order of magnitude improvements in energy efficiency over a Tensor Cores-based accelerator while improving performance by at least $4\times$ and as much as $15\times$ depending on the model and on-chip buffering capacity. Optionally, *Mokey* can be used as a memory compression assist for any other accelerator, transparently stashing wide floating-point or fixed-point activations or weights into narrow 4-bit indexes. *Mokey* proves superior to prior state-of-the-art quantization methods for Transformers.**

I. Introduction

Creating machines that can "understand" our language and "interact" with us as we interact with each other has always been a dream that motivated many and captured the imaginations of even more. Transformer models have demonstrated remarkable success towards this goal by tackling many natural language understanding tasks. Recently, Open-AI's GPT-3 [1], with its roughly 175 billion parameters, and Google's Switch transformer with its 1.6 trillion parameters [2], proved that attention-based models — given enough parameters — can perform tasks that they have not been explicitly trained for. For example, with zero-shot learning [1], [3], a model that is pre-trained for language modeling on copious amounts of text (including Common Crawl's Petabyte datasets [4]) could answer various questions, including solving a mathematical equation or writing an HTML script.

Accepted at the 49th IEEE/ACM International Symposium on Computer Architecture (ISCA '22) – DOI: 10.1145/3470496.3527438

The development trend of these NLP models shows that the more the parameters, the more powerful the model becomes. Unfortunately, this comes at a cost: More parameters require more storage, data transfers, and computation [5]. From 2018 with GPT [6] and BERT [7] with hundreds of millions of parameters to 2021 with the 1.6 trillion parameter Switch transformer, parameter footprint increased from hundreds of megabytes to terabytes. Computing hardware must meet their storage and compute needs and to do so within the constraints of each use case. As these models are massive and as the on-chip memory of modern silicon chips is limited, it is off-chip memory accesses that account for most of the execution time and energy consumption [8]–[10].

Transformer models also incur a quadratic growth in activation footprint when scaling the input sequence. For sequences of up to 128 tokens (e.g., GLUE dataset tasks [11]), buffering activations between layers requires anywhere between 1.5MB to 2MB depending on the model, layer, and dataflow. Keeping these activations on-chip is certainly possible and avoids the one to two orders of magnitude higher energy and latency costs of off-chip memory. However, when processing longer sequences such as multiple paragraphs or a book chapter, activation volume can easily exceed on-chip buffer capacities [12]–[15]. As an example, Figure 1 shows the total footprint of activations and weights as a function of sequence length for BERT-Large. When the sequence length exceeds 512 tokens, activations dominate total memory footprint.

Model compression reduces model size improving overall efficiency. Recent model compression works fall under three general classes: *Pruning* which forces some weights or activations to zero [16]–[23] combined with "zero-aware" memory encoding. *Knowledge distillation* distills a larger, "teacher" model into a smaller "student" model [24]–[30]. Lastly, *quantization* where the parameters and/or activations are quantized to shorter bit-widths [9], [31]–[35]. By default transformers use single-precision or 16b floating-point values.

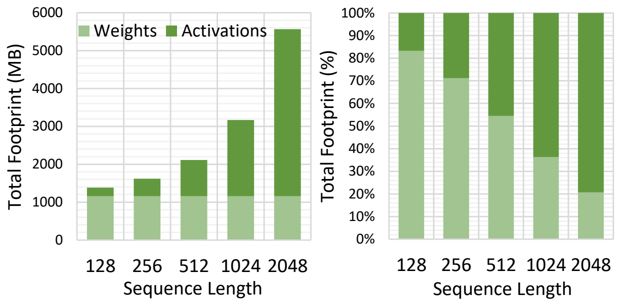


Fig. 1: BERT-Large: Weight and Activation Memory Footprint.

Except for I-BERT [33] which quantizes all values to 8b fixed-point, quantization methods still require floating-point or mixed fixed-point and floating-point units. In addition, all aforementioned compression methods require fine-tuning — further training for a few epochs over a previously trained model — to compensate for lost accuracy. Fine-tuning is expensive, requires access to sufficient system resources [36] and worse to datasets that might be unavailable to end-users. Having to search for optimal hyperparameters and quantization per task further exacerbates the cost of fine-tuning. GOBO is a post-training compression method for weights only that requires no fine-tuning [9]. GOBO quantizes the vast majority of weights to 3b or 4b indexes into a dictionary of 8 or 16 representative floating-point values (centroids). The very few remaining outlier values remain as-is (in single-precision 32b floating-point/FP32 or FP16). Further, GOBO still requires floating-point units. Ultimately, a quantization method that 1) does not require fine-tuning, 2) quantizes weights and activations, and 3) would require only fixed-point units, would improve overall energy efficiency and execution time performance. This is the goal of this work.

Quantizing activations presents additional challenges compared to quantizing weights only. First, since weights are statically known, quantization is done *off-line* using iterative centroid selection algorithms. The runtime and energy overheads of these algorithms are prohibitive for the *runtime* calculated activations. Quantizing activations calls for lightweight, *online* methods. Second, the value range of activations is much wider than those for weights (see Table IV) [31]–[34]. Most quantization methods still require 8 bits per activation or forego activation quantization altogether.

This work introduces *Mokey*, a *post-training* transformer model compression method for activations and weights that reduces memory traffic and computation costs. The key characteristics of *Mokey* are:

1) It does not require fine-tuning.
2) It quantizes *all* weights and activations into 16-entry dictionaries of *16 bit fixed-point* centroids storing *all* weights and activations as narrow indexes in memory.
3) It performs most computations directly on 4b dictionary indexes *without* having to map them to their corresponding 16b centroids. This is *Mokey*'s most innovative aspect.
4) It uses fixed-point arithmetic yielding 16b fixed-point activations.
5) It quantizes the 16b fixed-point output activations into 4b indexes on-the-fly.
6) It generates all dictionaries *off-line* using a light-weight method utilizing linear transformations over a pre-generated, model independent "Golden Dictionary".
7) The Golden Dictionary is symmetric around zero requiring only half of the entries to be stored.
8) It uses two dictionaries per tensor, one for the bulk of the values and another for outliers.
9) Optionally, *Mokey* can be used as a compression plug-in to *any* other accelerator.

In more detail, *Mokey* exploits the skewed, bell-shaped (Gaussian) distribution of values naturally occurring in transformer models: most of values are densely populated around their mean (in a small range) and a small fraction of values (covering a wider range) are outliers.

A key innovation in *Mokey* is that it adjusts the centroids in each dictionary to fit the original tensor's Gaussian distribution *and* an *exponential function*. This is only possible because, for the limited range of values used in transformers, a Gaussian distribution can be approximated well by an exponential function. It is this property that allows *Mokey* to replace 16- or 32-bit floating-point multiply-accumulate operations with narrow, 3-bit, integer additions (see Section II).

Additionally, *all* the dictionaries are derived from a reference "Golden" Dictionary which is created using a non-linear quantization scheme based on Agglomerative clustering [37]. The Golden Dictionary is independent of specific models and layers and the quantization method is shared across all models. This results in a lightweight post-training quantization process that simply applies a linear transformation of the Golden Dictionary so that it fits the distribution of each tensor (as described by the mean and standard deviation).

Optionally, *Mokey* can be used just for memory compression to only compress values to 4b indexes when storing them to higher and more energy demanding levels of the memory hierarchy while expanding them to 16b fixed-point or FP16 floating-values when requested by the lower memory hierarchy levels or directly by the processing units.

We evaluate *Mokey* over several FP16 pre-trained NLP models and highlight the following experimental findings:

- *Mokey* quantizes model parameters (weights, embeddings) and activations to 4b indexes with an average of 1.5% outliers for parameters and 4.5% outliers for activations while maintaining model accuracy.
- *Mokey*'s compute units consume $2.7\times$ less energy and are $1.6\times$ smaller than FP16 Tensor Cores units and boost energy efficiency by one to two orders of magnitude based on different configurations.
- *Mokey* as a memory-compression-only method for FP16 models when deployed over a Tensor-cores based accelerator magnifies on-chip capacity and reduces off-chip traffic by $4\times$ improving overall energy efficiency by $11\times$.

II. Mokey Quantization

It has been repeatedly shown that the values flowing through state-of-the-art attention-based NLP models form bell-shaped distributions per layer [9], [38], [39].

Mokey's quantization exploits these naturally occurring distributions that *all* weights and *most* activations exhibit. In contrast to past quantization methods that quantize each tensor separately, *Mokey* uses a "Golden Dictionary" approach which proceeds as follows:

Step 1: Golden Dictionary Generation: *Mokey* first generates and quantizes randomly generated, bell-shaped distributions with a mean of zero and a standard deviation of one. This step is performed once and is independent of the model under consideration. The resulting dictionary of centroids becomes *Mokey*'s "Golden Dictionary". In this work, this dictionary contains 16b fixed-point values.

Step 2: Per Tensor (Activation and Weights) Dictionary Generation: *Mokey* performs *a profiling run* of the model collecting samples of the activation tensors. It uses the activation tensor samples and the statically known weight tensors to scale and shift the Golden Dictionary to best fit each tensor. These linear transformations utilize the target tensor's mean and standard deviation. *Mokey* produces two dictionary sets per layer, one per input tensor — depending on the layer the input is a (weight, activation) or an (activation, activation) tensor pair. Per tensor, *Mokey* splits the value range into two subsets: 1) the Gaussian subset (G) contains the majority of values (more than 98% for weights and 95% for activations) which covers values near the mean. 2) The Outliers (OT) which are the rest of the values that are scattered in a much wider range. *Mokey*'s profiling-based approach for activation works well; while individual activation values vary with the input, their overall distribution per layer remains relatively unaffected. To validate this approach profiling runs use a single randomly selected batch containing 8 input samples (however, runs with even fewer input samples proved enough). Efficacy measurement runs use a non-overlapping set of input samples from the validation set of each target task. While prior quantization methods use an iterative per tensor approach [9], [31], [40], *Mokey*'s quantization method, does so only during the Golden Dictionary generation. Per tensor dictionary generation is non-iterative, resulting in a much faster overall process.

Step 3: Pre-Encoding of Weights: *Mokey* then encodes all weight tensors as indexes to their dictionaries (done offline).

At this point, the model is ready for inference.

A. Inference: Runtime Encoding/Decoding of Activations

During runtime *Mokey* produces 16b fixed-point output activations. These are converted to short indexes to the respective dictionaries prior to storing to memory. Since the dictionaries have few entries, this is a lightweight process that requires only a few runtime comparisons. Recall that in the original model, each output activation typically entails several hundreds of multiply-accumulate operations. As Section II-D will explain, the *Mokey* accelerator performs most of the computation directly on indexes and does not need to convert input activations into their 16b centroids.

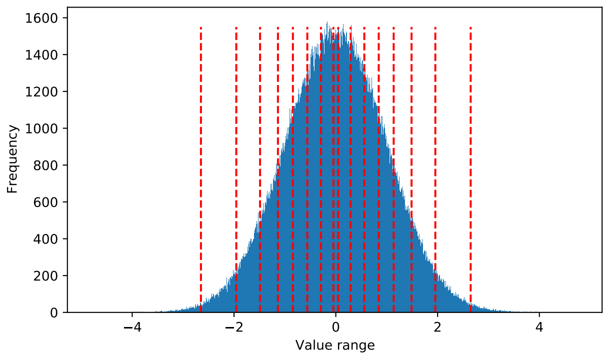


Fig. 2: Generating Golden Dictionary from a Random Gaussian Distribution Using Agglomerative Clustering.

The rest of this section explains the dictionary generation process and the exponential function approximation used to replace the original floating-point multiply-accumulate operations with narrow fixed-point operations comprising mostly additions of dictionary indexes.

B. Generating the Golden Dictionary (GD)

The core of *Mokey*'s dictionary generation method is Agglomerative Clustering (AC), a bottom-up approach which initially considers each value as a separate cluster and that proceeds to iteratively merge the closest clusters to reduce overall cluster count to a desired target value. In contrast to K-means, the method of choice in several prior works, Agglomerative Clustering is not affected by the initial cluster selection and results in higher accuracy in the quantized model. However, applying AC even when done offline given that weight or activation tensors can contain nearly a million values is impractical since AC requires $\mathcal{O}(n^2)$ memory and $\mathcal{O}(n^3)$ runtime. *Mokey* overcomes this challenge by exploiting the morphological similarity of the value distributions across tensors to apply AC using a "Golden Dictionary" which it then scales to best fit each individual tensor. In essence, this approach applies AC run on a representative distribution to *estimate* AC's clustering for the per tensor distributions. The process is simple: first, generate a random Gaussian distribution with 50,000 samples with a mean of zero ($m = 0$) and a standard deviation of one ($s = 1$). Then apply AC method on this distribution to produce the quantization dictionary. To create the Golden Dictionary, we repeat this process and compute an average over quantization dictionaries. Figure 2 shows the histogram for an example generated distribution and the resulting Golden Dictionary. In this work, Golden Dictionary is generated once and is reused across all models studied.

C. Generating Per Tensor Dictionaries

Mokey fits the Golden Dictionary (GD) to each tensor by first determining the mean (m) and the standard deviation (s)

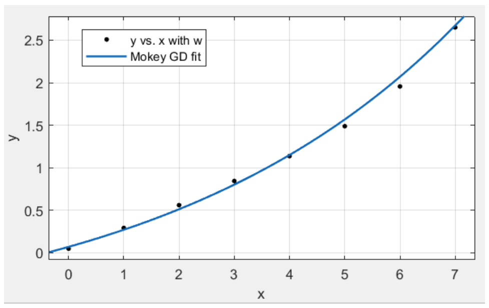


Fig. 3: Fitting an Exponential Curve to Golden Dictionary.

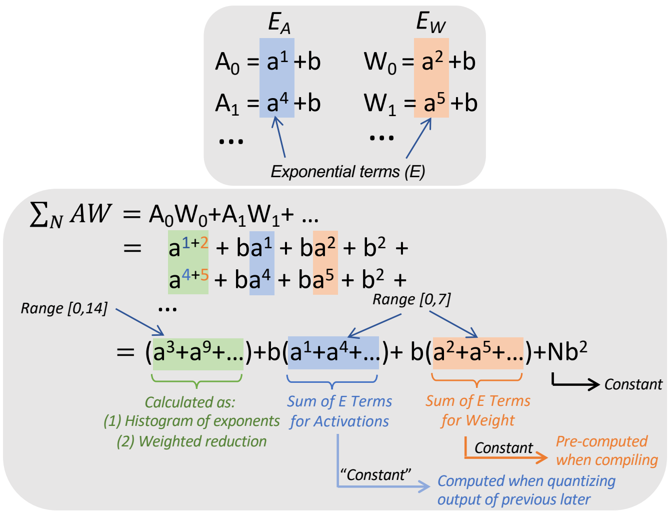
of the tensor's values. These two parameters are constant for weights and are estimated using profiling for activations. A simple linear transformation of GD is all that is needed: $GD \times s + m$. This is done once per tensor using profiling.

Dictionary size affects overall accuracy. In general, the more entries a dictionary has, the better it represents the original tensor distribution and the better it preserves overall model accuracy. However, larger dictionaries incur higher overheads. To better balance dictionary overheads while preserving accuracy *Mokey* uses two small dictionaries per tensor. The first "Gaussian" dictionary quantizes the bulk of the values, whereas the second "Outlier" dictionary maps the few remaining values of much larger magnitude. To best explain how *Mokey* chooses the respective ranges, it is best to first discuss how *Mokey* manages to map what were originally floating-point values and operations into fixed-point values and operations. Unique to *Mokey*, the bulk of these operations manipulate narrow fixed-point dictionary *indexes* instead of their respective *centroids*.

D. Integer Exponents for Compute Acceleration

Mokey quantizes weights and activations to a few unique values. For clarity, this section assumes that there is a single dictionary of 16 values which all follow a Gaussian distribution with a narrow range near its mean (outlier handling is described in Section II-E). With this quantization scheme, there are just 256 (16×16) possible multiplication products instead of the 2^{32} possible products with the original FP16 model. This represents an opportunity to replace the native FP16 multiply-accumulate (MAC) operations, with counting per possible product (integer histogram), followed by a final multiplication of the integer counts with the product of their respective centroids. However, we anticipate that having 256 counters per processing element will be impractical due to high energy and latency overhead compared to an FP16 MAC unit.

We observe that a Gaussian distribution, provided that we are only interested in values near its mean, can be approximated well with an exponential function where values are represented as $a^{int} + b$, where int is an integer index and a and b are appropriately chosen constants. As we explain here, this approximation enables *Mokey* to reduce what otherwise would have been 256 possible products into just 15 possible exponent sums, making a histogram-based processing MAC unit practical. Figure 4 illustrates, informally, how *Mokey* decomposes MAC operations into exponent counting (histogram) for quantized indexes and some constants. In this simplified example, all values are assumed to be positive and the layer's mean and standard deviation are assumed to be zero and one, respectively.



E_A
E_W
$A_0 = a^1 + b$
$A_1 = a^4 + b$
$W_0 = a^2 + b$
$W_1 = a^5 + b$
Exponential terms (E)
$\sum_N AW = A_0W_0 + A_1W_1 + \dots$
$= a^{1+2} + ba^1 + ba^2 + b^2 +$
$a^{4+5} + ba^4 + ba^5 + b^2 +$
Range [0,14]
Range [0,7]
$= (a^3+a^9+\dots)+b(a^1+a^4+\dots)+b(a^2+a^5+\dots)+Nb^2$
Constant
Calculated as:
(1) Histogram of exponents
(2) Weighted reduction
Sum of E Terms for Activations
Sum of E Terms for Weight
Constant
Pre-computed when compiling
"Constant"
Computed when quantizing output of previous later

Fig. 4: Calculating output activations without having to expand the quantized input activations and weights to their corresponding centroids.

In more detail: Since the Golden Dictionary is symmetrical around 0 and for a limited range, we first fit an exponential curve to one-half of the Golden Dictionary. The other half holds the opposite values. We used MATLAB's curve-fitting toolbox [41] to fit a curve of the form $a^{int} + b$ where a, b are the parameters that we are fitting and int is a list of integers from zero to half of the dictionary size. Since we have more values near zero and the density of the values decreases as we go sideways, we apply a weighting scheme during curve fitting to emphasize more densely populated ranges. The weighting scheme uses a unit weight for the outer bin, and doubles the weight for the bins as we move towards zero. For 4 bit quantization, this results in 16 bins which are symmetrical around zero: 8 are positive and 8 are negative with identical magnitudes as their corresponding positive bin. We fit the $GD = a^{int} + b$ curve on these 8 positive values where $a = 1.179$, $b = -0.977$, where int is an integer in range of $[0, 7]$ and the fitting weights are in $[2^7, 2^0]$ range. Figure 3 shows the resulting exponential function (curve) and the original Golden Dictionary entries (dots).

In the resulting 4 bit quantization, the most significant bit is the sign of the value (0: positive , 1: negative) and the following 3 bits are the index. For instance, if the quantized representation is $0b1011$ the corresponding value would be $-(a^{\mathbf{3}} + b)$. We emphasize that a and b are fitted to the Golden Dictionary and this process is performed only *once* prior to scaling the resulting dictionary to each tensor.

Computing using Dictionary Indexes instead of Dictionary Centroid Values: We can now exploit that our values fall on an exponential curve to simplify multiplication. At a high-level the method takes advantage of property $a^i \times a^j = a^{i+j}$. After approximating the dictionary with values on an exponential curve, we are guaranteed that when we are multiplying two input tensors, input values are of the form $\theta(a^{int}+b) \times s + m$ where $theta$ is the sign, int a 3b Golden Dictionary index, and s and m are constants, respectively, the standard deviation and the mean of the tensor.

For clarity, let us assume that the layer operates on a weight W and an activation A tensor. The respective values would then be of the form: $\theta_W(a^{int_W}+b) \times s_W + m_W$ and $\theta_A(a^{int_A}+b) \times s_A + m_A$ where θ_A, θ_W are signs, int_W, int_A are integer indexes, s_W, s_A are standard deviations and m_W, m_A are means for the corresponding weight and activation tensors, respectively.

The weight and activation values can be simplified to:

$$\begin{aligned} A &= \theta_A(a^{int_A}+b) \times s_A + m_A \\ &= \theta_A s_A.a^{int_A} \underbrace{+\theta_A s_A.b + m_A}_{\mu_A} = \theta_A s_A.a^{int_A} + \mu_A \end{aligned} \tag{1}$$

$$\begin{aligned} W &= \theta_W(a^{int_W}+b) \times s_W + m_W \\ &= \theta_W s_W.a^{int_W} \underbrace{+\theta_W s_W.b + m_W}_{\mu_W} = \theta_W s_W.a^{int_W} + \mu_W \end{aligned} \tag{2}$$

where a is a fitted parameter which is constant for all tensors across all layers and models, a sign (θ), int_A and int_W are dictionary indexes, S_A and S_W are per tensor scalars scaling factors, and μ_A and μ_W are shifts.

Each output activation is a sum of activation times weight products, which given the form of the input values can be expanded into a sum of four terms:

$$\begin{aligned} \sum_{i=0}^{n-1} A_i.W_i &= \sum_{i=0}^{n-1} \left(\theta_{A_i} s_A.a^{\text{int}A_i} + \mu_{A_i}\right)\left(\theta_{W_i} s_W.a^{\text{int}W_i} + \mu_{W_i}\right) \\ &= \underbrace{s_A.s_W \sum_{i=0}^{n-1} \left(\theta_{A_i}\theta_{W_i} a^{\text{int}A_i + \text{int}W_i}\right)}_{SoI} \\ &\quad \underbrace{+ s_A \sum_{i=0}^{n-1} \left(\mu_{W_i}\theta_{A_i} a^{\text{int}A_i}\right)}_{SoA} \\ &\quad \underbrace{+ s_W \sum_{i=0}^{n-1} \left(\mu_{A_i}\theta_{W_i} a^{\text{int}W_i}\right)}_{SoW} + \underbrace{\sum_{i=0}^{n-1} (\mu_{A_i}.\mu_{W_i})}_{PoM} \end{aligned} \tag{3}$$

The first term (SoI) is the sum of $a^{\text{int}_{A_i} + \text{int}_{W_i}}$. Assuming 4b quantization (3b indexes plus 1b signs), $int_A + int_W$ will be in range of $[0+0, 7+7] = [0, 14]$ which given the quantization contains 15 unique values. Since the exponents are limited to 15 values we can first add int_A and int_W and count how many times each exponent occurs (count how many a^0, $a^1 ... a^{14}$ we have overall) then with 15 MAC operations (occurrences $\times a^0 + ...+$occurrences $\times a^{14}$) we can compute the SoI term.

The second term (SoA) can be decomposed into two terms as per Eq. 4. The first term (SoA_1) is a summation of activations with respect to the sign of both weight and activation. This term is computed during computation similar to SoA, but it needs smaller counters as the range of activation is limited to $[0, 7]$. The second term (SoA_2) is the sum of the input activations for this layer. This term can be computed while the output of previous layer is being quantized so that is available in advance of this layer.:

$$\begin{aligned} SoA &= s_A \sum_{i=0}^{n-1} \left(\mu_{W_i}\theta_{A_i} a^{\text{int}A_i}\right) = s_A \sum_{i=0}^{n-1} \left((\theta_{W_i} s_W.b + m_W)\theta_{A_i} a^{\text{int}A_i}\right) \\ &= \underbrace{s_A.s_W.b \sum_{i=0}^{n-1} \left(\theta_{W_i}\theta_{A_i} a^{\text{int}A_i}\right)}_{SoA_1} + \underbrace{s_A.m_W. \sum_{i=0}^{n-1} \left(\theta_{A_i} a^{\text{int}A_i}\right)}_{SoA_2} \end{aligned} \tag{4}$$

The third term (SoW), similar to SoA is decomposed into two term as per Eq 5. The first term (SoW_1) is the summation of the weights with respect to the sign of both weight and activation, and it is computed similar to (SoA_1). The second term (SoW_2) is a summation over weights and is a constant that can be computed statically before inference.

$$\begin{aligned} SoW &= s_W \sum_{i=0}^{n-1} \left(\mu_{A_i}\theta_{W_i} a^{\text{int}W_i}\right) \\ &= s_W \sum_{i=0}^{n-1} \left((\theta_{A_i} s_A.b + m_A)\theta_{W_i} a^{\text{int}W_i}\right) \\ &= \underbrace{s_W.s_A.b \sum_{i=0}^{n-1} \left(\theta_{A_i}\theta_{W_i} a^{\text{int}W_i}\right)}_{SoW_1} + \underbrace{s_W.m_A. \sum_{i=0}^{n-1} \left(\theta_{W_i} a^{\text{int}W_i}\right)}_{SoW_2} \end{aligned} \tag{5}$$

The last term (PoM) is decomposed in Eq. 6 to 4 terms. The first term is a summation of the sign of multiplication (PoM_1) which is computed during inference. The second and the third terms are the summation of activations' sign (known when quantizing last layer) and weights' sign (known before inference) and the last term is a constant.

$$\begin{aligned} PoM &= \sum_{i=0}^{n-1} (\mu_{A_i}.\mu_{W_i}) = \sum_{i=0}^{n-1} \left((\theta_{A_i} s_A.b + m_A)(\theta_{W_i} s_W.b + m_W)\right) \\ &= \underbrace{s_A.s_W.b^2 \sum_{i=0}^{n-1} (\theta_{A_i}\theta_{W_i})}_{PoM_1} + \underbrace{s_A.m_W.b \sum_{i=0}^{n-1} (\theta_{A_i})}_{PoM_2} \\ &\quad \underbrace{+ s_W.m_A.b \sum_{i=0}^{n-1} (\theta_{W_i})}_{PoM_3} + \underbrace{n.m_A.m_W}_{PoM_4} \end{aligned} \tag{6}$$

In conclusion computation can proceed as follows: 1) Add the indexes (3 bit values) for a weight and an activation we want to multiply. 2) XOR the sign bit of weight and activation and based on that result, 3) increment or decrement the occurrences tables for SoI (15 entries), SoA_1 (8 entries), SoW_1 (8 entries), and PoM_1 (1 entry). 4) After filling the occurrences tables (which typically will require thousands of cycles given the tensor sizes), multiply each count with its corresponding value ($a^0 ... a^{15}$) and accumulate all products into a single value. 5) The final values are multiplied by their coefficients and added with pre-computed terms (which at this point are constants) ($SoA_2, SoW_2, PoM_{2,3,4}$) producing the final output activation.

The output activation can then be quantized by comparing it against the dictionary for the output activation tensor. In practice, we find that 16-entry dictionaries prove sufficient.

Values are quantized to a 4b indexes which are a 3b index into the stored dictionary entries and a 1b sign.

E. Outliers

Outliers (OT) are exceedingly rare and represent less than 2% of weights and 5% of activations (varies per model and task). However, they do span over a disproportionally large value range. To quantize outliers, we adapt the Golden Dictionary approach.

We could naïvely use a common dictionary for all values. Starting with the 16 centroid dictionary for the G values, we found that to support outliers, we need to widen the index range to $int = 45$. Encoding the index to such a dictionary would require 6(index)+1(sign)=7 bits per value. We use a two-prong approach to limit index lengths to 4b. First, we use a separate outlier dictionary. Second, we take advantage of the exceedingly rare occurrence of outliers to store all values, outlier or Gaussian, using a 4b index. A separate, space- and access-efficient list of pointers explicitly identifies which indexes correspond to outliers (all others index to the Gaussian dictionary). Section III-A describes the encoding.

Outliers require different handling during computation as they do not fit under the same exponents (0 to 7) as the Gaussian values. For any product where at least one of the values is an outlier, *Mokey* converts the values into their centroids and performs a multiply-accumulate. Since outliers are exceedingly rare, this is a rare occurrence (less than 4% of the multiplications in BERT). However, unless further action is taken, the centroids would be FP16 numbers necessitating FP16 capable MAC hardware. Fortunately, *Mokey* is further enhanced to require only fixed-point values and hardware.

F. Integer Computation Throughout

Integer compute units are faster and more energy-efficient than floating-point units. *Mokey* maps what were originally floating-point computations to the fixed-point (integer) domain. The conversion to fixed-point arithmetic is done during profiling. The process starts after the mean, standard deviation, the dictionary (bins) that we are going to use for outliers and the few constants (SoW, PoM) have been derived. To convert these parameters to integer, *Mokey* first needs to set the total bit-width (b) and to compute the number of fractional bits per layer ($frac$). The number of fractional bits ($frac$) needed is:

$$frac = b - \text{ceil}\left(\log_2(\max - \min)\right) \tag{7}$$

where max and min are the maximum and minimum values that appear for the layer. The float (fl) to fixed-point(fx) mapping then is:

$$fx = \text{round}(fl \times 2^{frac})/2^{frac} \tag{8}$$

G. Summary

In summary, *Mokey* quantized an input model as follows: First, it generates the Golden Dictionary for quantization. This step is done once and the output is reused across *all* models. Second, it quantizes the model's parameters and embeddings. On average, these statically known values consist of about 98.5% G values and 1.5% outliers. G values are stored in the form of exponents (4-bit) and outliers are stored as small (16 entry–4-bit) dictionary indexes where the dictionary entries are 16b fixed-point values. A list of pointers identifies which indexes are to the outlier dictionary. Third, *Mokey* performs a run over a randomly selected batch of inputs to generate the dictionaries for the activations. The profiling is to find the mean, standard deviation, and the outlier bins that activations are using. The dictionaries and other constants per tensor are stored along with the model. The space needed for this metadata pales in comparison with the size of the respective tensors; a G dictionary (8×16b entries), an OT dictionary (16×16b), plus a few constants and a list of outlier pointers.

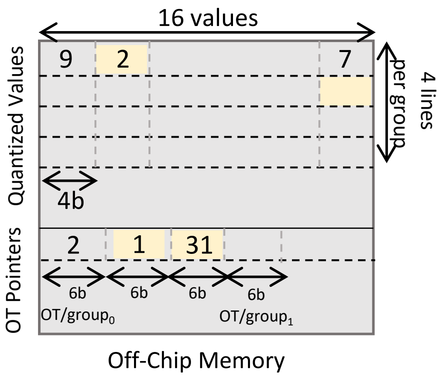


Fig. 5: DRAM layout

III. Mokey Memory Layout and Processing Units

This section describes the memory encoding and processing units of *Mokey*. Combined they reduce memory traffic and footprint while performing most computations using narrow fixed-point arithmetic.

A. Off-Chip Memory Layout

To take advantage of *Mokey*'s quantization for reducing off-chip memory traffic and footprint, we need to map Gaussian and outlier values in a DRAM-friendly data container. Per tensor, Gaussian and outlier values are represented as 4b indexes. Since outliers are rare, *Mokey* stores all values using 4b indexes as shown in Figure 5. An extra list of pointers identifies which of those indexes are outliers. The pointers are encoded as follows: Conceptually, the regular "Quantized value" 4b index array is split into groups of 64, 4b indexes. To identify those indexes that are outliers, the "OT Pointers" list first stores outlier count per group, followed by a list of 6b indexes marking their relative position within the group. In the example shown, $group_0$ has two outliers at positions 1 and 31. This container is DRAM-friendly as accesses to the two sub-arrays will be done through two streams, each proceeding through the respective areas sequentially with the bulk of accesses going to the "Quantized Values" area. For simplicity, at an appropriate level of the on-chip hierarchy the values can be expanded to 5b (dictionary selection/1b, sign/1b, centroid index/3b) indexes. This facilitates single stream accesses per tensor.

B. Processing Units

Mokey hardware accelerator enables compute units to directly operate on the quantized indexes using narrow fixed-point processing elements (PEs) for the bulk of computation (Gaussian "G" values). Only for products where at least one of the operands (weight or activation) is an outlier ("OT"), *Mokey* PEs convert the values into their respective int16 centroids. This requires a simple lookup to the respective dictionary.

Tile architecture: Figure 6 shows the architecture of processing unit of the *Mokey* hardware accelerator. As shown on the left, the accelerator comprises an array of 8 cascaded Gaussian PEs (GPEs) which share an outlier/post-processing circuitry (OPP). At peak, each GPE$_i$ processes a group of compressed activations ($A_{0\ldots7}$) and the corresponding group of compressed weights ($W_{0\ldots7}$). The GPEs process the Gaussian-distributed inputs only; the outliers are sent to the OPP unit for lookup and accumulation. The OPP unit is activated only when an outlier is received, or by the end of the computation for post-processing. At a high-level, most activation and weight pairs will use Gaussian values and will be processed by the GPEs that accumulate the partial sums described in Section II-D. The occasional pairs containing outliers are processed in the OPP. Since these are rare, they are processed one at a time for simplicity and area-efficiency. After the full tensors have been processed, a final step uses the OPP to accumulate and scale the various summations into the output 16b activation. The unit processes activations and values that have been encoded in 5b (1b dictionary index, 1b sign, 3b dictionary index).

The GPE is detailed in Figure 6 (middle). In the core of each GPE, four individually sized Counter Register-Files (CRFs) accumulate the SoI, SoA_1, SoW_1, and PoM_1 summations (Section II-D). The organization of the CRF units is illustrated in Figure 6 (right). Each line (addressed by `wAddr`) can be incremented or decremented using $\texttt{up}/\overline{\texttt{down}}$, if `wEn` is asserted. The read address, `rAddr` is used to scan the register-file content during post-processing using the post-processing address `ppAddr`. The multiplication output sign ($\theta_A\theta_W \equiv \texttt{sgnA} \oplus \texttt{sgnA}$) is used to increment/decrement all CRFs. SoI is counted using a 15×8 CRF, and is addressed with $A + W \equiv \texttt{idxA+idxW}$. SoA_1 is counted using a 8×8 CRF, and is addressed with $A \equiv \texttt{idxA}$. SoW_1 is counted using a 8×8 CRF, and is addressed with $W \equiv \texttt{idxW}$. Finally, PoM_1 is counted using a 1×8 CRF (a single byte). As the post-processing is performed only once for each layer, its timing is less critical. Thus, the post processing is performed serially. Only one of the summations (SoI, SoA_1, SoW_1, and PoM_1) is selected for post processing as `sData`, and is selected by the control signal `sumSel`.

Scheduling outliers processing: Since it is exceedingly rare but possible that more than one GPE receive an outlier simultaneously, the processing of outliers must be scheduled: the lowest index GPE that contains an outlier is selected for processing; all other GPEs with outliers send a hold signal back through the input channel (`hldA`, and `hldW`, to hold the activations and weights, respectively). Each GPE generates a local outlier-is-present signal $\texttt{isOtlCur} = \texttt{isOtlA} \vee \texttt{isOtlW}$. A serial, cascaded leading-one-detector across all GPEs detects the first GPE with outlier ($\texttt{isOtlCur} = 1$). The selected "leading" GPE with an outlier asserts its `otlSel` signal ($\texttt{otlSel} = 1$) forcing the `otlSel` signals to be deasserted ($\texttt{otlSel} = 0$) for all other GPEs. The one-hot-encoded `otlSel` is used to route the corresponding indexes to the OPP (routing is done via per GPE AND gates and an OR level shared across all GPEs).

The Outlier/Post-Processing (OPP) unit: First, when an outlier has been detected, where the `isOtl` signal is asserted, the OPP uses its G/OT-LUT to retrieve the outlier value and performs a MAC operation. Second, in Post-processing where `sData` will hold one of the summations (SoI, SoA_1, SoW_1, and PoM_1). A single PE will be selected iteratively using the selector `pePESel`. The G/OT-LUT will operate as a G-LUT and produce the bases of the summations. Finally, the bases from the G-LUT and the summations from the GPEs will be multiplied and accumulated.

Output Activation Quantizers: Prior to storing them in memory, the output activations (OA) of each layer are quantized using the prepared OT and G dictionaries. Figure 7 shows the organization of the quantization units. An output activation OA is compared with every centroid from both the G and the OT dictionaries. Since the dictionary values are *sorted*, and assuming 16 values per dictionary, the output of the comparators will be a 32b vector of 0s followed by 1s. A leading-one detector drives two 32-to-1 multiplexers, respectively, selecting the two corresponding 16b centroids CL and CH: the one corresponding to the leading 1 position and the one before it or the same if the position happens to be the first one. To determine the nearest to the OA centroid of the two, OA is subtracted from each of CL and CH, and the two differences are compared to find the smaller of the two identified by the signal Ci. The relative position of this centroid is then encoded as a 5b index. If written off-chip, a controller packs them to the format of Figure 5.

C. Using Mokey For Memory Compression Only

Optionally, *Mokey* can be used just as a memory compression method where the values are transparently converted to fixed-point 16b or (FP16 if desired) when written or read from an appropriate level in the memory hierarchy. In this case, when reading values, lookup tables can convert the indexes into their corresponding centroids. Quantizers placed just before the memory level pack the activation values into indexes to dictionaries. When the compression targets an on-chip buffer, the activations are best encoded with 5b (1b dictionary OT or G, 1b sign, 3b index), whereas when writing to off-chip memory, the *Mokey* off-chip memory format would improve space efficiency. Using the less efficient 5b representation on-chip avoids the overheads of creating the OT Pointer metadata and the need for two access streams per tensor.

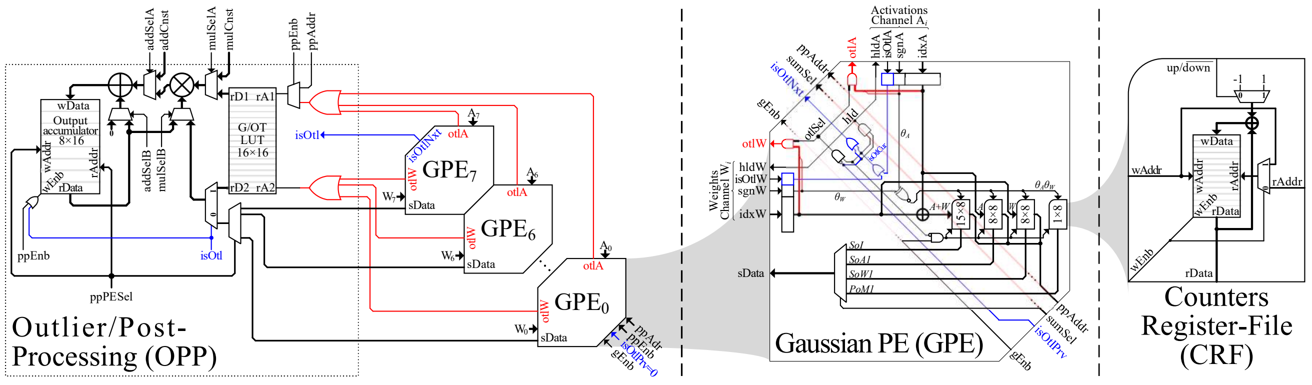


Fig. 6: Architecture of *Mokey* Hardware Accelerator.

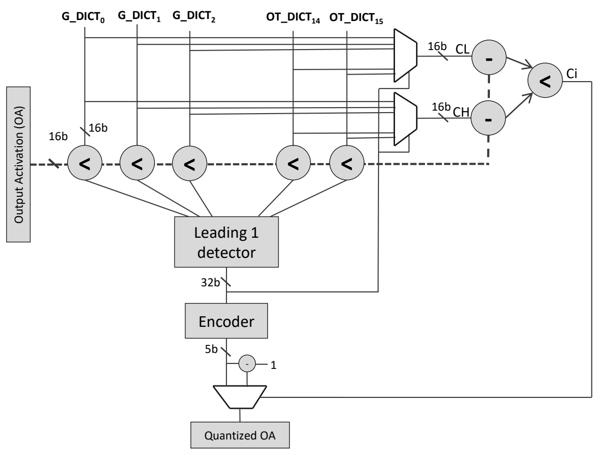


Fig. 7: Output Quantization Engine.

IV. Evaluation

A. Model Task Performance

We first evaluate the effect of *Mokey* quantization on *task performance* over various models, tasks, and datasets. Task performance refers to how *well* the model performs each task. The metric used is task-dependent. We show that *Mokey* maintains task performance without requiring fine-tuning.

We applied *Mokey* on the following models: DeBERTa [42], a recent model from Microsoft that is in the top-ranked models on GLUE [11] benchmark, on Facebook's RoBERTa [43] which is used widely used in sentiment analysis, and on models from Google's BERT family of models [7] which powers the search engine to show more related results in search queries. The pre-trained models were obtained from the Hugging Face Model hub [44]. The Golden Dictionary was created by generating a random normal distribution and applying Agglomerative Clustering using the SciKit-Learn library [45]. Our main benchmark to evaluate model accuracy is the MNLI task in the GLUE dataset, as it is the most sensitive task to quantization and has the most comprehensive set for language inference in the dataset [9], [39]. Furthermore, MNLI trained models are publicly available. Where possible, however, we also study STS-B and SQuAD.

BERT: Bidirectional Encoder Representations from Transformers (BERT) [7] is a well-known and widely used transformer model for NLP applications such as similarity detection, sentiment analysis, and question answering. We apply *Mokey* on BERT-Base (12 Encoders, 110M parameters) and BERT-Large (24 Encoders, 340M parameters) on MNLI (The Multi-Genre Natural Language Inference), STS-B (Semantic Textual Similarity Benchmark) from GLUE benchmark, and SQuADv1 [46] (Stanford Question Answering Dataset) task.

RoBERTa: A Robustly Optimized BERT Pretraining Approach (RoBERTa) [43] shares the same architecture as BERT but benefits from an optimized training recipe that leads to higher accuracy. We evaluate *Mokey* on RoBERTa-Large using MNLI, STS-B and SQuAD v1 tasks. RoBERTa-Large has 24 Encoder layers and a total of 340M parameters.

DeBERTa: Decoding-enhanced BERT with disentangled attention (DeBERTa) [42] is a state-of-the-art transformer model that increases the accuracy of Transformers by decoupling the position of the tokens from their context embedding. We evaluate DeBERTa-XL (48 Encoder layers, 750M parameters) on the MNLI task since trained MNLI models are publicly available.

Table I reports the resulting task performance under the "Score" columns. The "FP Score" column reports the task performance of the original FP32/FP16 models, whereas the other two score columns report task performance when we quantize weights alone or weights and activations. The higher the score, the better the model performs. The change in task performance is also listed under the "Err" columns. Overall, *Mokey* is within 1% of the FP baseline and closer for the most recent DeBERTa. The table also lists the fraction of outlier values in weights (W OT%) and activations (A OT%). In the worst case, outliers account for just 2.5% of the overall values. In some applications, there may be enough on-chip buffering to store the activations as-is. Accordingly, the table also reports task performance when *Mokey* is applied only to weights. Task performance is, as expected, slightly better and, in some cases, even better than the FP baseline. Such variations in task performance are normal. The outlier fraction is also noticeably lower for the weights, which is also expected as the activations

TABLE I: The Effect of *Mokey* Quantization on Task Performance (see text for definition.)

Model Configuration				Weight only Quant.			Weight + Activation Quant.		
Model	Task	Metric	FP Score	W OT%	Score	Err	A OT%	Score	Err
BERT-Base	MNLI	Acc-m	84.44	1.6	84.80	-0.36	4.5	84.22	0.22
BERT-Large	MNLI	Acc-m	86.65	1.51	86.39	0.26	4	85.69	0.96
BERT-Large	STS-B	Spearman	90.25	1.51	90.12	0.13	2.5	89.51	0.74
BERT-Large	SQuAD	F1	93.15	1.54	93.17	-0.02	1.7	92.22	0.93
RoBERTa-Large	MNLI	Acc-m	90.58	1.48	90.38	0.20	4.1	89.81	0.77
RoBERTa-Large	STS-B	Spearman	92.41	1.48	92.25	0.16	4.4	91.52	0.89
RoBERTa-Large	SQuAD	F1	93.56	1.48	93.25	0.31	2.9	92.58	0.98
DeBERTa-XL	MNLI	Acc-m	91.75	1.2	91.78	-0.03	4.3	91.18	0.57

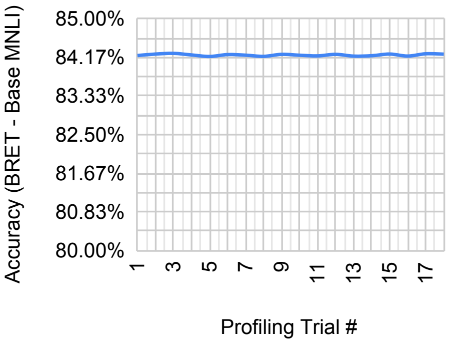


Fig. 8: Profiling effect on accuracy.

TABLE II: Area, Cycle Count and Energy for BERT-Base.

	Compute		BERT-Base/512KB buffer	
Architecture	Units	Area (mm^2)	Cycle Count	Energy (J)
FP16 Tensor Cores	2048	16.1	167M	0.36
FP16 GOBO	2560	15.9	52M	0.17
Mokey	3072	14.8	29M	0.09

exhibit a much larger range, as corroborated by the use of longer datatypes for activations in prior quantization work.

Our experiments show profiling has almost no effect on the accuracy: Figure 8 shows the accuracy after profiling model multiple times with different random samples of the training set and the result of profiling is almost identical each time. The reasons behind this are *I)* In our method, the bins get exponentially wider when the index (a^{index}) gets larger. This means the bins that will contain outliers are wide and capture an extremely wide range of values. *II)* weights in the model act as a regularizer for the activations and shape the output distribution. Also, transformer-based models have layer normalization and softmax which limits the range of values.

B. Hardware Evaluation Methodology

As our baseline, we use a spatial 2048 FP16 Tensor Cores-based accelerator [47], [48] and we sweep a range of possible on-chip buffer capacities. We use this baseline to demonstrate that *Mokey* compression alone can benefit an existing accelerator and to also demonstrate that a design using the *Mokey* processing units can outperform it. We also compare with GOBO [9]. The dataflow for all designs is optimized to minimize the number of off-chip transactions.

Cycle counts are measured using a custom cycle-accurate simulator that uses DRAMSIM3 [49] to model DRAM transactions for a DDR4-3200 dual-channel main memory. The simulator has been tested with various microbenchmarks and verifies the correctness of the simulated output. Memories are divided into banks to meet the target access time and bandwidth. Area and power for the on-chip memories are modeled using CACTI [50].

We use *post-layout* energy and area estimates. All designs are implemented in Verilog. Major arithmetic units are instantiated from Synopsys DesignWare Building Block IP Library [51]. For synthesis, we use Synopsys' Design Compiler (ver. N-2017.09) [52] with a 65nm TSMC technology node targeting 1 GHz frequency and for placement and routing, we use Cadence Innovus (ver. 20.13) [53]. Power consumption is estimated in Innovus using the post-layout netlist with an activity factor generated by Intel ModelSim.

C. Mokey Acceleration

The baseline architecture uses FP16 — an FP32 baseline which is more common today, would have resulted in higher benefits for *Mokey*. Table II reports area used by their processing elements and their count for the baseline Tensor Cores, GOBO (refer to section V) and *Mokey*. Since the area of each *Mokey* processing element (PE) is smaller than either those of Tensor Cores or GOBO, *Mokey* can afford to pack more elements within less area. Moreover, since *Mokey* stores values using short indexes, it does not need wide memories to supply its units. For example, the *Mokey* PE is 39% smaller compared to a tensor-core unit with an equivalent compute-capability (MAC/cycle). This superior processing power per area and energy vs. the baseline is an additional advantage of *Mokey*.

Comparison with Tensor Cores: We first compare *Mokey* with the baseline Tensor Cores accelerator reporting execution time performance and energy efficiency. All measurements are normalized over the corresponding baseline configuration. Tensor Cores and *Mokey* use the same compute area and the same buffer capacity – total chip area for *Mokey* is smaller as it uses narrower buffers and interconnect. Figure 9 reports the execution time in cycles for the baseline configurations.

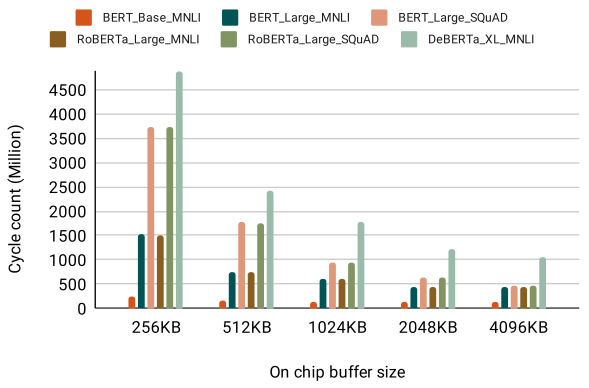


Fig. 9: Baseline Accelerator: Inference Cycle Counts.

As expected, using larger on-chip buffers decreases execution time as 1) it allows for more overlap between off-chip accesses and on-chip compute, and 2) reduces the number of off-chip transactions due to higher reuse.

Figure 10 shows that *Mokey* reduces execution time compared to the baseline. Although *Mokey* has some overheads for computing outliers and for the post-processing stage, on average it outperforms the baseline by $11\times$ when the buffers are small and by $4.1\times$ faster for the large 4MB buffers.

Figure 11 also reports the resulting relative energy efficiency for the *Mokey* accelerator. Even for the 4MB on-chip buffer configurations, the *Mokey* hardware accelerator boosts energy efficiency by $13\times$. For the 256KB configurations *Mokey* is $78\times$ more energy efficient.

Advantages of *Mokey* Hardware Accelerator: Table III shows breakdowns of area, performance, and energy for Tensor Cores and *Mokey* with BERT Large on SQuAD. **1) Area:** Since *Mokey* PEs are 39% smaller, more can fit in the same compute area. Furthermore, since *Mokey* computes directly on quantized indexes, it requires on-chip buffers with significantly narrower data *interfaces* and *interconnects* with area and energy benefits. For example, due to the narrower interface, *Mokey*'s 1MB buffers use as much area as the 256KB buffers of Tensor Cores. However, in this evaluation, we did *not* use this to *Mokey*'s advantage, e.g., by introducing more units. Instead we used an *iso-compute-area* and *iso-buffer-capacity* constraint and showed that *Mokey* outperforms Tensor Cores even under this *worse* for *Mokey* constraint. If we were to account for this advantage of *Mokey*, combined the $4\times$ reduction in memory area and the reduction of values from 16b to 5b, would translate into a nearly $13\times$ amplification of on-chip memory capacity. Section IV-D shows that the *Mokey* accelerator is still better than the Tensor Cores even when the latter uses *Mokey* just for on- and off-chip memory compression. Processing with Tensor Cores requires converting the indexes into the corresponding centroids via lookup tables.

2) Performance: *Mokey*'s on-chip compression allows more values to be stored on-chip and increases the overlap of compute and memory transactions. For example, with the 256KB on-chip buffer, *Mokey* increases compute/memory overlap from 27% to 58%. Also, since *Mokey* can fit more weights and activations on-chip it can better reuse values and reduce the overall off-chip traffic. **3) Energy:** *Mokey* compute units are extremely energy efficient. *Mokey*, despite having $1.5\times$ more compute units, it burns $2\times$ less energy as they perform mostly counting on narrow fixed-point indexes.

Comparison with GOBO: Figure 12 shows that *Mokey* is faster than the GOBO accelerator. The figure shows *Mokey*'s execution time normalized over GOBO with the same on-chip buffer size. Whereas GOBO stores activations in FP16, *Mokey* quantizes them to 4b. This magnifies on-chip buffer capacity, boosting performance for tasks with longer sequence lengths (e.g., SQuAD). With larger on-chip buffers, *Mokey* remains faster albeit at a lesser degree since the source of performance benefits shifts more to the difference in peak compute capability. Another advantage of *Mokey* is that its PEs are more efficient as they perform fixed-point operations vs. GOBO's FP16 PEs. This is shown in Figure 13 which reports the overall energy efficiency of *Mokey* over GOBO. *Mokey* consumes less energy and is $9\times$ more energy efficient compared to GOBO with smaller buffers. The energy efficiency remains high at $2\times$ even with the 4MB on-chip buffers.

D. *Using Mokey for Memory Compression Only*

We consider using *Mokey* only as compression mechanism (weights and activations) over the Tensor Cores baseline. Larger on-chip buffers store more values on-chip, resulting in 1) better overlap between computation and memory transactions, and 2) a reduction in DRAM transactions.

***Mokey* Off-Chip Compression:** In the first set of experiments, *Mokey* compresses values off-chip only. When the values are transferred to the chip, they get expanded into FP16 centroids. Figure 14 shows performance normalized over the corresponding baseline configurations that do not use *Mokey* compression. The average speedup for a small 256KB buffer is about $3.9\times$ and it improves when the on-chip buffer gets larger to $4.3\times$ for a 4MB buffer. These models are mostly memory bound, and with the smaller on-chip buffers, compute units consume the data faster than DRAM transactions resulting in idle time. These configurations most of the time cannot overlap computation with off-chip memory transactions. As the on-chip capacity grows larger, computation and off-chip access overlap becomes increasingly frequent.

In general, off-chip compression reduces DRAM energy consumption by a factor of $4\times$ reducing overall relative energy (Figure 15). With the 256KB on-chip buffer, on average 82% of total energy is due to memory transactions. This is less with the larger on-chip buffers as they access DRAM less. With the 4MB on-chip buffer, memory transactions account for 53% of energy consumption. Regardless, *Mokey* off-chip compression improves energy efficiency of the 256KB configuration by $11\times$ and by $7.8\times$ for the 4MB configuration.

***Mokey* On-Chip Compression:** In this application, *Mokey* stores 5-bit indexes in on-chip buffers and expands the

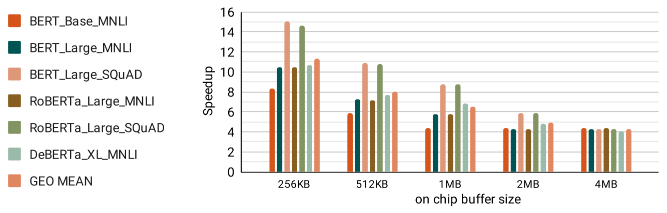


Fig. 10: *Mokey* Accelerator Speedup over Tensor Cores.

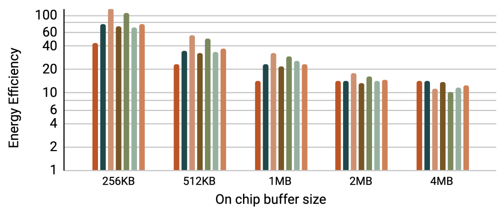


Fig. 11: *Mokey* Energy Efficiency over Tensor Cores.

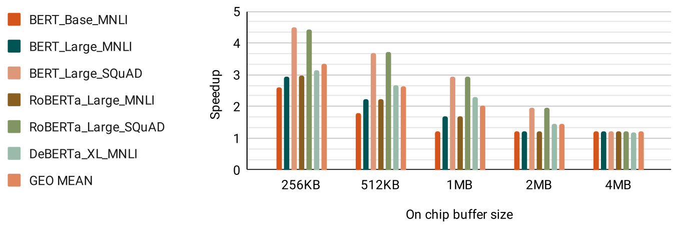


Fig. 12: *Mokey* Accelerator Speedup over GOBO.

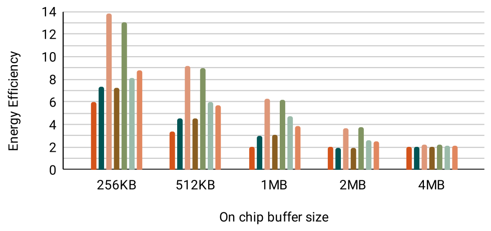


Fig. 13: *Mokey* Energy Efficiency over GOBO.

TABLE III: Area, Performance and Energy breakdown for Tensor Cores and *Mokey* for BERT Large on SQuAD.

	256KB on-chip buffer		512KB on-chip buffer		1MB on-chip buffer	
	Tensor Cores	*Mokey*	Tensor Cores	*Mokey*	Tensor Cores	*Mokey*
Area (mm^2)						
On-chip Buffer Area	13.2	4.7	16.8	8.0	24.7	14.6
Compute Area	16.1	14.8	16.1	14.8	16.1	14.8
Total Chip Area	**30.7**	20.5	34.5	23.9	42.7	**30.8**
Performance						
Memory Transfer Cycles	3690 M	226 M	1730 M	151 M	938 M	108 M
Compute Cycles	60 M	55 M	60 M	55 M	60 M	55 M
Total Cycles	3734 M	249 M	1772 M	163 M	953 M	109 M
Compute/Memory Avg. Overlap %	26.7%	57.7%	29.0%	77.0%	76.5%	98.2%
Energy (J)						
Off-chip Memory Energy	5.79	0.35	2.72	0.24	1.47	0.17
On-chip Memory Energy	0.1	0.01	0.1	0.01	0.11	0.02
Compute Energy	0.95	0.48	0.95	0.48	0.95	0.48
Total Energy	6.84	0.84	3.77	0.73	2.53	0.67

values once they are requested by compute units. In Figure 14 *(OC+ON)* set of bars report the corresponding speedup. Compressing data allows us to store more weights and activations, maximize reuse and significantly reduces the number of DRAM transactions. The smaller the on-chip capacity, the higher the potential benefits from on-chip compression. In transformer models, activations grow quadratically with respect to sequence length. BERT-Large and RoBERTa-Large on the SQuAD task used a sequence length of 384 tokens, while for other model/tasks use a sequence length of 128. This explains why these two models on SQuAD task benefit the most from on-chip compression in the smaller buffer sizes. As the on-chip buffer gets larger, eventually, all activations would fit on-chip and the weights need to be loaded only once from off-chip. With *Mokey* compression, the on-chip buffer size at which this becomes possible is smaller.

Energy reduction with on-chip memory compression exhibits a similar trend as with off-chip as shown in Figure 15. Energy reduction is significant with the smaller on-chip buffers as *Mokey* boosts their effective capacity reducing off-chip accesses. With the 256KB buffer *Mokey* improves energy efficiency by $54\times$ and by $8\times$ with the 4MB buffer.

V. Related Work

This section discusses how *Mokey* compares to state-of-the-art model compression methods for Transformer models.

Integer/Floating-Point Quantization: Q8BERT quantized weights and activations to 8-bit fixed-point. It requires fine-tuning to reduce quantization error and FP32 compute cores for some layers such as Softmax [32]. I-BERT also quantizes weights and activations to 8 bits, but by leveraging approximations for nonlinear functions it uses fixed-point throughout [33]. Table IV compares Q8BERT and I-BERT with *Mokey*. Mokey is a post-training method and can be applied on out-of-the-box models without requiring access to datasets or systems that are capable of training these models. *Mokey* reduces activation and weight footprint by approximately $2\times$

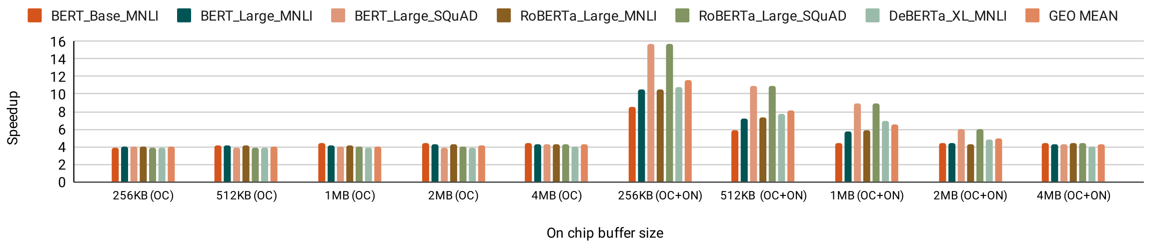


Fig. 14: Tensor Cores speedup w/ *Mokey* Memory compression for Off-chip only (OC) and Off- and On-chip (OC+ON) traffic.

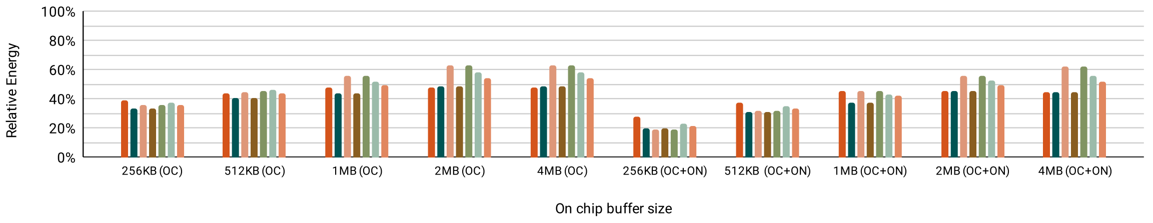


Fig. 15: Tensor Cores Relative Energy w/ *Mokey* Memory compression for Off-chip only (OC) and Off- and On-chip (OC+ON).

TABLE IV: Comparing Various Quantization Methods for BERT-Base on the MNLI task. *INT Comp* indicates if the computation is in fixed-point domain. *Compression Ratio* reports the total memory footprint reduction.

BERT-Base MNLI	Parameters (bit)	Activations (bit)	Accuracy (m)	Error	INT Comp	Post-Training	Compression Ratio
FP32 Baseline	32	32	84.44	-	✗	✗	1×
Q8BERT [32]	8	8	83.75	0.69	✓	✗	4.0×
I-BERT [33]	8	8	84.12	0.32	✓	✗	4.0×
Q-BERT [31]	4	8	83.89	0.55	✗	✗	6.9×
GOBO [9]	3	32	83.76	0.68	✗	✓	4.1×
TernaryBERT [34]	2	8	83.30	1.14	✗	✗	10.8×
Mokey	4	**4**	84.22	**0.22**	✓	✓	7.9×

compared to Q8BERT and I-BERT, has higher accuracy, and performs all computations in fixed-point with most using 3b.

Dictionary-Based Quantization: Q-BERT is a group-wise dictionary-based quantization method [31]. It divides the parameters of each layer into groups (typically 128) each of which it quantizes to a dictionary of 4 to 16 representative values. Q-BERT quantizes activations to 8b. It requires fine-tuning. GOBO is a post-training dictionary-based quantization method that quantizes weights only to 3b or 4b (activations remain in FP16 or FP32) [9]. GOBO uses an iterative centroid selection method similar to k-means. Table IV compares Q-BERT and GOBO with *Mokey*. Compared to Q-BERT, *Mokey* requires no fine-tuning, reduces footprint for activations more, and has better accuracy. Compared to GOBO, *Mokey* quantizes both weights and activations, achieves higher accuracy, and replaces floating-point arithmetic with narrow fixed-point operations. *Mokey* quantizes layers based on a Golden Dictionary which is faster than GOBO's iterative process.

Pruning, Distillation and NAS: Model pruning, Distillation and Neural Architecture Search (NAS) can reduce model size and are orthogonal to quantization methods such as *Mokey*. TernaryBERT uses a combination of knowledge distillation and quantization to compress BERT [34]. It quantizes weights to 2b and activations to 8b. TernaryBERT's extreme quantization comes at the expense of a 1.14% drop in accuracy, and the distillation process requires training the model while searching for suitable hyperparameters. SpAtten prunes less effective cascade heads and tokens from attention layers [23]. HAT is a NAS method which alters model architecture to best fit a target hardware configuration [54].

VI. Conclusion

The bulk of the operations and data transfers in state-of-the-art NLP models are due to multiply-accumulate operations (MACs) over floating-point values. *Mokey* introduces a unique quantization that maps the original floating-point value space into a few fixed-point values which are a small subset of a larger fixed-point value space. *Mokey* chooses this subset so that *all* its values fit under some exponential curve. It is this property that enables *Mokey* to replace MACs over floating-point tensors, transforming them into counting of 4b centroid indexes without even having to map to the centroid values themselves — at the end, a few scaling operations using fixed-point arithmetic map the final result into the target fixed-point value space. Since *Mokey* adjusts the exponential curve to best fit each layer, it does not restrict the set of possible centroid values. We expect that future work will expand and improve upon quantization methods such as *Mokey*. These are methods that carefully select not only the target value space, but also a subset within it to maximize cost reduction opportunities.

Acknowledgement

This work was supported by the NSERC COHESA Strategic Research Network and an NSERC Discovery Grant. The University of Toronto maintains all rights to the technologies described.

References

[1] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[2] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *CoRR*, vol. abs/2101.03961, 2021.

[3] M.-W. Chang, L. Ratinov, D. Roth, and V. Srikumar, "Importance of semantic representation: Dataless classification," in *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, p. 830–835, AAAI Press, 2008.

[4] C. C. nonprofit 501(c)(3) Organization, 2021.

[5] R. Desislavov, F. Martínez-Plumed, and J. Hernández-Orallo, "Compute and energy consumption trends in deep learning inference," *arXiv preprint arXiv:2109.05472*, 2021.

[6] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2019.

[8] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study on optimizing transformers," *Proceedings of Machine Learning and Systems*, vol. 3, 2021.

[9] A. H. Zadeh, I. Edo, O. M. Awad, and A. Moshovos, "GOBO: quantizing attention-based NLP models for low latency and energy efficient inference," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, pp. 811–824, IEEE, 2020.

[10] P. Ganesh, Y. Chen, X. Lou, M. A. Khan, Y. Yang, H. Sajjad, P. Nakov, D. Chen, and M. Winslett, "Compressing large-scale transformer-based models: A case study on bert," *Transactions of the Association for Computational Linguistics*, vol. 9, p. 1061–1080, Sep 2021.

[11] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, OpenReview.net, 2019.

[12] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are rnns: Fast autoregressive transformers with linear attention," in *International Conference on Machine Learning*, pp. 5156–5165, PMLR, 2020.

[13] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," *arXiv preprint arXiv:2006.04768*, 2020.

[14] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, *et al.*, "Big bird: Transformers for longer sequences.," in *NeurIPS*, 2020.

[15] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[16] J. Mao, H. Yang, A. Li, H. Li, and Y. Chen, "Tprune: Efficient transformer pruning for mobile devices," *ACM Trans. Cyber-Phys. Syst.*, vol. 5, apr 2021.

[17] S. Kim, S. Shen, D. Thorsley, A. Gholami, W. Kwon, J. Hassoun, and K. Keutzer, "Learned token pruning for transformers," *arXiv preprint arXiv:2107.00910*, 2021.

[18] F. Lagunas, E. Charlaix, V. Sanh, and A. M. Rush, "Block pruning for faster transformers," *arXiv preprint arXiv:2109.04838*, 2021.

[19] A. Parnami, R. Singh, and T. Joshi, "Pruning attention heads of transformer models using a* search: A novel approach to compress big nlp architectures," *arXiv preprint arXiv:2110.15225*, 2021.

[20] J. Li, R. Cotterell, and M. Sachan, "Differentiable subset pruning of transformer heads," *arXiv preprint arXiv:2108.04657*, 2021.

[21] D. Peer, S. Stabinger, S. Engl, and A. Rodriguez-Sanchez, "Greedy layer pruning: Decreasing inference time of transformer models," *arXiv preprint arXiv:2105.14839*, 2021.

[22] K. Shim, I. Choi, W. Sung, and J. Choi, "Layer-wise pruning of transformer attention heads for efficient language modeling," *arXiv preprint arXiv:2110.03252*, 2021.

[23] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 97–110, IEEE, 2021.

[24] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *5th Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS*, 2019.

[25] G. Aguilar, Y. Ling, Y. Zhang, B. Yao, X. Fan, and C. Guo, "Knowledge distillation from internal representations," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 7350–7357, 2020.

[26] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, "Training data-efficient image transformers & distillation through attention," in *International Conference on Machine Learning*, pp. 10347–10357, PMLR, 2021.

[27] R. Masumura, N. Makishima, M. Ihori, A. Takashima, T. Tanaka, and S. Orihashi, "Hierarchical transformer-based large-context end-to-end asr with large-context knowledge distillation," in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5879–5883, IEEE, 2021.

[28] S. Mukherjee and A. H. Awadallah, "Distilling bert into simple neural networks with unlabeled transfer data," *arXiv preprint arXiv:1910.01769*, 2019.

[29] L. Liu, H. Wang, J. Lin, R. Socher, and C. Xiong, "Mkd: a multi-task knowledge distillation approach for pretrained language models," *arXiv preprint arXiv:1911.03588*, 2019.

[30] Y. Jiang, B. Sharma, M. Madhavi, and H. Li, "Knowledge distillation from bert transformer to speech transformer for intent classification," *arXiv preprint arXiv:2108.02598*, 2021.

[31] S. Shen, Z. Dong, J. Ye, L. Ma, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "Q-bert: Hessian based ultra low precision quantization of bert.," in *AAAI*, pp. 8815–8821, 2020.

[32] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, "Q8bert: Quantized 8bit bert," *5th Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS*, 2019.

[33] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-bert: Integer-only bert quantization," *arXiv preprint arXiv:2101.01321*, 2021.

[34] W. Zhang, L. Hou, Y. Yin, L. Shang, X. Chen, X. Jiang, and Q. Liu, "Ternarybert: Distillation-aware ultra-low bit bert," *arXiv preprint arXiv:2009.12812*, 2020.

[35] I. Chung, B. Kim, Y. Choi, S. J. Kwon, Y. Jeon, B. Park, S. Kim, and D. Lee, "Extremely low bit transformer quantization for on-device neural machine translation," *arXiv preprint arXiv:2009.07453*, 2020.

[36] O. Sharir, B. Peleg, and Y. Shoham, "The cost of training NLP models: A concise overview," *CoRR*, vol. abs/2004.08900, 2020.

[37] M. L. Zepeda-Mendoza and O. Resendis-Antonio, *Hierarchical Agglomerative Clustering*, pp. 886–887. New York, NY: Springer New York, 2013.

[38] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. K. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware," *ACM Computing Surveys*, vol. 52, p. 1–39, May 2019.

[39] M. A. Gordon, K. Duh, and N. Andrews, "Compressing bert: Studying the effects of weight pruning on transfer learning," *arXiv preprint arXiv:2002.08307*, 2020.

[40] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2016.

[41] MATLAB, *version 9.9.0 (R2020a)*. Natick, Massachusetts: The MathWorks Inc., 2020.

[42] P. He, X. Liu, J. Gao, and W. Chen, "Deberta: Decoding-enhanced bert with disentangled attention," *arXiv preprint arXiv:2006.03654*, 2021.

[43] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[44] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "Huggingface's transformers: State-of-the-art natural language processing," *ArXiv*, vol. abs/1910.03771, 2019.

[45] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[46] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," *arXiv preprint arXiv:1606.05250*, 2016.

[47] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," *CoRR*, vol. abs/1804.06826, 2018.

[48] L. Durant, O. Giroux, M. Harris, and N. Stam, "Nvidia developer blog," May 2017.

[49] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "Dramsim3: A cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.

[50] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, 01 2009.

[51] Synopsys Inc., *DesignWare® Building Block IP User Guide*, December 2016. Ver. M-2016.12.

[52] Synopsys Inc., *Design Compiler® User Guide*, September 2017. Ver. N-2017.09.

[53] Cadence Design Systems Inc., *Cadence Innovus User Guide*, January 2021. Ver. 20.13.

[54] H. Wang, Z. Wu, Z. Liu, H. Cai, L. Zhu, C. Gan, and S. Han, "Hat: Hardware-aware transformers for efficient natural language processing," *arXiv preprint arXiv:2005.14187*, 2020.