A. Quantization Methods

A.1. Symmetric and Asymmetric Quantization

Symmetric and asymmetric quantization are two different methods for uniform quantization. Uniform quantization is a uniform mapping from floating point $x \in [x_{\min}, x_{\max}]$ to b-bit integer $q \in [-2^{b-1}, 2^{b-1} - 1]$. Before the mapping, input x that does not fall into the range of $[x_{\min}, x_{\max}]$ should be clipped. In asymmetric quantization, the left and the right side of the clipping range can be different, i.e., $-x_{\min} \neq x_{\max}$. However, this results in a bias term that needs to be considered when performing multiplication or convolution operations (Jacob et al., 2018). For this reason, we only use symmetric quantization in this work. In symmetric quantization, the left and the right side of the clipping range must be equal, i.e., $-x_{\min} = x_{\max} = \alpha$, and the mapping can be represented as Eq. 1.

A.2. Static and Dynamic Quantization

There is a subtle but important factor to consider when computing the scaling factor, S. Computing this scaling factor requires determining the range of parameters/activations (i.e., α parameter in Eq. 1). Since the model parameters are fixed during inference, their range and the corresponding scaling factor can be precomputed. However, activations vary across different inputs, and thus their range varies. One way to address this issue is to use dynamic quantization, where the activation range and the scaling factor are calculated during inference. However, computing the range of activation is costly as it requires a scan over the entire data and often results in significant overhead. Static quantization avoids this runtime computation by precomputing a fixed range based on the statistics of activations during training, and then uses that fixed range during inference. As such, it does not have the runtime overhead of computing the range of activations. For maximum efficiency, we adopt static quantization, with all the scaling factors fixed during inference.

B. Error Term of Eq. 3

As one can see, the polynomial approximation of Eq. 3 exactly matches the data at the interpolating points (x_j, f_j). The error between a target function $f(x)$ and the polynomial approximation $L(x)$ is then:

$$|f(x) - L(x)| = \left| \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)\dots(x - x_n) \right|, \quad (16)$$

where ξ is some number that lies in the smallest interval containing $x_0, ..., x_n$. In general, this error reduces for large n (for a properly selected set of interpolating points). Therefore, a sufficiently high-order polynomial that interpolates a target function is guaranteed to be a good approximation for it. We refer interested readers to (Stewart, 1996) for more details on polynomial interpolation.

C. Experimental Details

C.1. Implementation

In I-BERT, all the MatMul operations are performed with INT8 precision, and are accumulated to INT32 precision. Furthermore, the Embedding layer is kept at INT8 precision. Moreover, the non-linear operations (i.e., GELU, Softmax, and LayerNorm) are processed with INT32 precision, as we found that keeping them at high precision is important to ensure no accuracy degradation after quantization. Importantly, note that using INT32 for computing these operations has little overhead, as input data is already accumulated with INT32 precision, and these non-linear operations have linear computational complexity. We perform Requantization (Yao et al., 2020) operation after these operations to bring the precision down from INT32 back to INT8 so that the follow up operations (e.g., next MatMuls) can be performed with low precision.

C.2. Training

We evaluate I-BERT on the GLUE benchmark (Wang et al., 2018), which is a set of 9 natural language understanding tasks, including sentimental analysis, entailment, and question answering. We first train the pre-trained RoBERTa model on the different GLUE downstream tasks until the model achieves the best result on the development set. We report this as the baseline accuracy. We then quantize the model and perform quantization-aware fine-tuning to recover the accuracy degradation caused by quantization. We refer the readers to (Yao et al., 2020) for more details about the quantization-aware fine-tuning method for integer-only quantization. We search the optimal hyperparameters in a search space of learning rate $\{5e{-}7, 1e{-}6, 1.5e{-}6, 2e{-}6\}$, self-attention layer dropout $\{0.0, 0.1\}$, and fully-connected layer dropout $\{0.1, 0.2\}$, except for the one after GELU activation that is fixed to 0.0. We fine-tune up to 6 epochs for larger datasets (e.g., MNLI and QQP), and 12 epochs for the smaller datasets. We report the best accuracy of the resulting quantized model on the development set as I-BERT accuracy.

C.3. Accuracy Evaluation on the GLUE Tasks

For evaluating the results, we use the standard metrics for each task in GLUE. In particular, we use classification accuracy and F1 score for QQP (Iyer et al., 2017) and MRPC (Dolan & Brockett, 2005), Pearson Correlation and Spearman Correlation for STS-B (Cer et al., 2017), and Mathews Correlation Coefficient for CoLA (Warstadt et al., 2019). For the remaining tasks (Dagan et al., 2005; Ra-

jpurkar et al., 2016; Socher et al., 2013; Williams et al., 2017), we use classification accuracy. For the tasks with multiple metrics, we report the average of them. Since there are two development sets for MNLI (Williams et al., 2017), i.e., MNLI-match (MNLI-m) for in-domain evaluation, and MNLI-mismatch (MNLI-mm) for cross-domain evaluation, and we report the accuracy on both datasets. We exclude WNLI (Levesque et al., 2012) as it has relatively small dataset and shows an unstable behaviour (Dodge et al., 2020).

C.4. Environment Setup for Latency Evaluation

We use TensorRT 7.2.1 to deploy and tune the latency of BERT-Base and BERT-Large models (both INT8 and FP32) on Google Cloud Platform virtual machine with a single Tesla T4 GPU, CUDA 11.1, and cuDNN 8.0.

We should also mention that the most efficient way of implementing BERT with TensorRT is to use NVIDIA's plugins (Mukherjee et al., 2019) that optimize and accelerate key operations in the Transformer architecture via operation fusion. Our estimates are that INT8 inference using NVIDIA's plugins is about 2 times faster than naïvely using TensorRT APIs. However, we cannot modify those plugins to support our integer-only kernels as they are partially closed sourced and pre-compiled. Therefore, our latency evaluation is conducted without fully utilizing NVIDIA's plugins. This leaves us a chance for further optimization to achieve our latency speedup relative to FP32 even more significant. As such, one could expect the potential for a further $\sim 2\times$ speed up with INT8 quantization.