Efficient 8-Bit Quantization of Transformer Neural Machine Language Translation Model

Aishwarya Bhandare[1] **Vamsi Sripathi**[1] **Deepthi Karkada**[1] **Vivek Menon**[2] **Sun Choi**[1] **Kushal Datta**[1] **Vikram Saletore**[1]

arXiv:1906.00532v1 [cs.LG] 3 Jun 2019

Abstract

In this work, we quantize a trained Transformer machine language translation model leveraging INT8/VNNI instructions in the latest Intel® Xeon® Cascade Lake processors to improve inference performance while maintaining less than 0.5% drop in accuracy. To the best of our knowledge, this is the first attempt in the industry to quantize the Transformer model. This has high impact as it clearly demonstrates the various complexities of quantizing the language translation model. We present novel quantization techniques directly in TensorFlow to opportunistically replace 32-bit floating point (FP32) computations with 8-bit integers (INT8) and transform the FP32 computational graph. We also present a parallel batching technique to maximize CPU utilization during inference. Overall, our optimizations with INT8/VNNI deliver 1.5X improvement over the best FP32 performance. Furthermore, it reveals the opportunities and challenges to boost performance of quantized deep learning inference and establishes best practices to run inference with high efficiency on Intel CPUs.

1. Introduction

The Transformer model using self-attention mechanism has recently achieved the state of the art accuracy in language translation (Vaswani et al., 2017). Compared to its predecessors, this sequence transduction model circumvents recurrent or long-short memory (LSTM) neural cells and exploits multi-headed attention mechanism to capture global dependencies between input and output word sequences. Since its first use in machine translation, the multi-headed attention mechanism has shown tremendous promise in speech recognition (Chiu et al., 2017), generative language modeling (Guu et al., 2017), machine reading comprehension (Hu et al., 2017), Language representation models (Devlin et al., 2018) and other natural language processing workloads.

The growing presence of machine language translation services and tools (Microsoft, 2018), (Google, 2018), (AWS, 2018) and (LingoTek, 2019) to name a few, clearly shows that machine translation inference is an important workload. Quantization is a technique to improve the performance of inference workloads by using lower precision data types (8-bit, 4-bit or 2-bit integers) in place of 32-bit floating point. The latest Intel ® Xeon® Cascade Lake processors include specialized vectorized neural network instructions (VNNI) to expedite quantized inference by fusing 64 8-bit multiply and add (FMA) operations into a single instruction (Fomenko, 2018). This means that the vectorized FMAs can be completed in fewer clock cycles than previous generation Intel ® Xeon® processors. As a result, 8-bit matrix multiplications (MatMuls) or quantized MatMuls execute faster on these platforms. This motivated us to explore the impact of VNNI on the performance of Transformer model inference. To the best of our knowledge, the Transformer model has not been quantized before. However, the impact of quantized MatMuls on the overall performance of Transformer inference was not known before this work as speedup between INT8 and FP32 MatMuls depend on the shape and size of the matrices involved.

Additionally, we want to minimize the drop in translation accuracy which can result due to the usage of reduced precision data types. In this work, our contributions include the following:

1. Quantized a trained FP32 Transformer model to INT8 to achieve $<$ 0.5% loss in state-of-the-art(SOTA) accuracy.
2. Improve inference performance by:
   (a) Optimizing quantized MatMuls for tensor shapes

*Equal contribution [1]Artificial intelligence products group, Intel Corporation [2]Work done while at Intel corporation. Correspondence to: Kushal Datta <kushal.datta@intel.com>, Vikram Saletore <vikram.saletore@intel.com>.

To be presented at the Joint Workshop on On-Device Machine Learning & Compact Deep Neural Network Representations, 36th International Conference on Machine Learning, Long Beach, California, 2019. Copyright 2019 by the author(s).

and sizes in the Transformer model
(b) Reducing overhead due to quantization operations in the Transformer model compute graph
(c) Optimizing input pipeline by ordering sentences by token length
(d) Implementing parallel execution of batches with increased inference throughput

The rest of the paper is organized as follows. In section 2, we describe prior work on quantization techniques for deep learning models. In section 3, we provide a brief description of the Transformer translation model. In section 4, we describe how we first quantize the graph to maintain accuracy and then in section 5 detail strategies to improve inference efficiency. In section 6, we present the overall improvement in inference performance.

2. Related work

Various compression techniques including quantization for deep neural network models have been well studied. (Jiaxiang Wu & Cheng, 2016) explored quantization with the objective of compressing and accelarating convolutional neural networks for mobile devices. (Hou & Kwok, 2018) proposed a ternerization scheme for quantization of weights to compress the neural network model, but didn't deal with quantization of activations.
(Mellempudi et al., 2017) introduced a sub 8-bit inference pipeline using weights constrained to +1, 0, -1 and 8/4-bit activations. By reducing precision of activations as well as weights of a trained model, they reduced the burden on the memory system but the inference performance improvement is not obtained. (Wu et al., 2018) proposed a method to quantize weights, activations, gradient computation, and accumulation to low-bitwidth integers. This low-precision integer dataflow is a comprehensive approach that can be applied for training and inference. Integer-based training and inference increases on-site learning capability of ASIC or FPGA, but extra operations added to reduce precision can degrade model performance.
(Park et al., 2018) proposed Value-aware quantization, by selectively quantizing small values in weights and activations, but retaining the large values in full precision during training and inference in very deep networks like ResNet-101 and DenseNet-121. Although this reduces the memory cost of activations, it does not leverage the speedup due to VNNI instructions.
Other quantization techniques have been adopted for Convolutional neural networks in (Courbariaux & Bengio, 2016),(Gysel et al., 2018), and (Migacz, 2017).
(Alom et al., 2018) proposed quantization techniques for recurrent neural networks using various thresholding mechanisms and performed evaluations on the IMDB dataset for the sentiment analysis task and observed promising performance results.
To the best of our knowledge, this is the first work where quantization has been applied to the Transformer network. Transformer networks differ from Convolutional Neural Networks and Recurrent Neural Networks in that, Transformer networks do not contain any convolutional or recurrent layers, but only the attention mechanism. This work is also the first to demonstrate the speedup observed in inference performance by leveraging VNNI for the Transformer Language Translation model.

3. Model Description

The architecture of the Transformer translation model with multi-headed attention is shown in figure 1 of (Vaswani et al., 2017). It has an encoder-decoder structure. The encoder maps the input sequence of tokens in source language to a sequence of latent representations. The decoder generates the translated sequence of tokens in target language from these representations. The decoder is auto-regressive which means that previously generated tokens are used to decode the next token using a while loop.
The model uses the scaled dot-product attention and multi-headed attention mechanisms. The equations Equation 1 and Equation 2 from (Vaswani et al., 2017) describe the fundamental computation in the model. It can be inferred from the equations that the primary operation in this model is a Matrix Multiplication (MatMul). It is also clear that the model contains the Softmax operation in between MatMuls. The Softmax operation in Equation 3 involves mapping the input to a probability distribution, and has a division operation. This would mean that computing a Softmax in a lower precision datatype would result in high accuracy loss as compared to computing it in full-precision FP32 datatype. In addition to Softmax, the graph has Layer Normalization(Lei Ba et al., 2016) layer in between any two layers(Vaswani et al., 2017). The Layer Normalization layer involves calculating the mean and variance of each layer, and normalizing the values of that layer. This involves operations like division, square and square root, which again are more accurate with a full-precision FP32 datatype over INT8 datatype. Thus, the entire computation graph of this model doesn't support low precision INT8 datatype. Parts of the graph need to be selectively transformed to work in low-precision, while keeping the remainder of the graph in FP32.

$$Attention(Q,K,V) = Softmax(\frac{QK^T}{\sqrt{d_K}})V \tag{1}$$

$$\begin{aligned} MultiHead(Q,K,V) = concatenate(&head_1, \\ &head_2,\ldots,head_h)W^O \end{aligned} \tag{2}$$
$$where, head_i = Attention(QW_1, KW_2, VW_3)$$

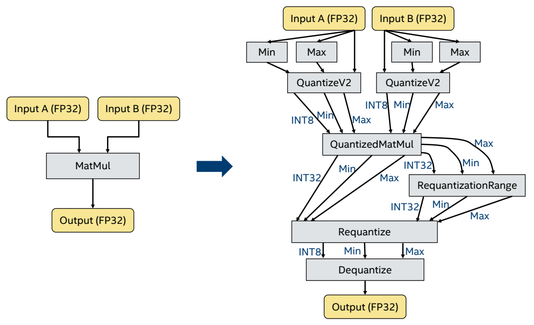


Figure 1. Schematic of Naïve Quantization for MatMuls in TensorFlow

$$Softmax(\phi_k) = \frac{\exp(\phi_k)}{\sum_j^c \exp(\phi_j)}, \tag{3}$$

Transformer model has two variations – base and big. The differences are in the number of weight dimensions and number of layers. For English to German translation, the base and big model achieve BLEU scores of 27.3 and 28.4 respectively. In this study, we retrained the base model and start with a BLEU score of 27.68, which is close to the state-of-the-art. (Wu, 2018).

4. Quantization with accuracy

Quantizing a model entails converting the FP32 tensors to a target lower precision integer data type as described in Equation 5 and then replace the FP32 operations with corresponding quantized operations.

$$scale = \frac{target}{Max - Min} \tag{4}$$

$$A_{quantized} = round((A_{float} - zero_{offset}) \cdot scale) \tag{5}$$

For example, to convert an FP32 tensor following Equation 5 to an unsigned INT8 tensor the simplest way is to map minimum (Min) and maximum (Max) values in the tensor to 0 and 255 respectively. Values between 0 and 255 represent the numerical distribution between Min and Max. This way we can represent an arbitrary magnitude of ranges to a lower precision datatype. The overhead of quantization is $O(N)$ as it requires linear scans to calculate Min and Max, and then transform the entire tensor. N here signifies the size of the tensor.

4.1. Naïve Quantization

The primary compute in neural networks is in the form of convolution and matrix multiplication operations. These operations benefit from quantization due to speed-up obtained from INT8/VNNI instructions. The Transformer LT model has MatMuls as seen in section 3, and MatMuls take up the largest portion of the computation as shown in Figure 7(a). Thus, MatMul is the first operation that must be quantized.

The naïve way of quantization is mapping the entire FP32 dynamic range of a tensor to INT8. The modification in the computation graph in TensorFlow as shown in Figure 1 involves replacing a quantizable operation (MatMul) with the corresponding quantized version of that operation(QuantizedMatMul) in the model graph, and passing quantized input tensors to that opeartion. The quantized tensors are obtained by passing the FP32 tensor through a QuantizeV2 operation to convert it to INT8. Operations to calculate Min and Max values of the tensor are also inserted before the QuantizeV2 node. The difference between quantized kernel of an operation and its FP32 version is that it executes ISA optimized INT8 instructions (including VNNI). For example, the QuantizedMatMul operation accepts the A matrix (signed INT8), the B matrix (unsigned INT8) and their corresponding Min/Max values. The result of the multiplication is accumulated as signed INT32 value. The RequantizationRange operation calculates the output Min and Max values from the INT32 result. This is fed into a Requantize operation which maps the INT32 result into an INT8 representation with a new Min and Max value. Since operations downstream in the computational graph may not accept INT8 datatype values, a Dequantize operation is required to convert the INT32 value back into an FP32 value as shown in Equation 6. Dequantization is also $O(N)$ in complexity.

$$A_{dequantized} = (Max - Min) \cdot (A_{quantized} - zero_{offset}) \tag{6}$$

This transform was done to all the MatMul operations in the graph. On running inference on the graph transformed using this method, it failed to emit a stop token at all, and consequently failed to achieve the less than 0.5% drop in the FP32 accuracy. This ended up deeming the naïve quantization approach inappropriate for this model. A closer investigation by visualizing the histograms of input tensor values show that most of them have a long-tailed numerical distribution.

If the entire range of the FP32 tensor was to be preserved, there would be a loss of precision due to multiple values being mapped to the same bin. Hence, quantization using absolute Min and Max can result in significant loss in accuracy. As a result naïve quantization is not a viable approach. In the next subsection, we explore other approximation methods based on divergence between probability distributions.

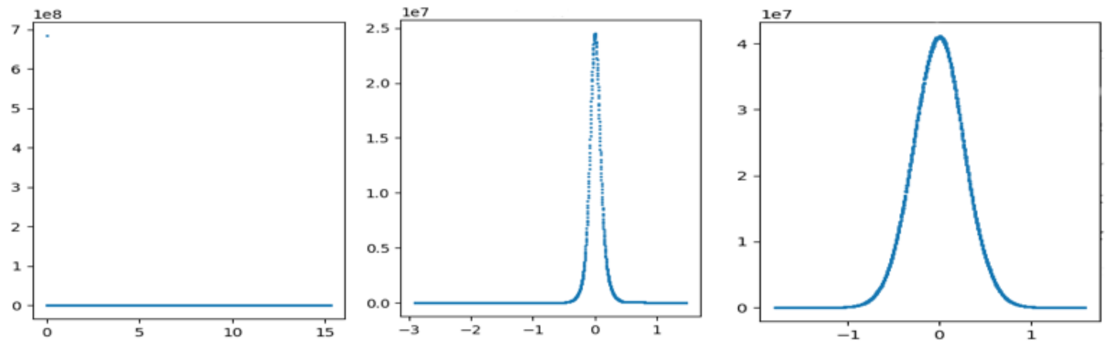


Figure 2. Tensors with sparse, narrow and Gaussian histograms in Transformer model

4.2. KL-Divergence for optimal saturation thresholds

One solution to preserve precision when quantizing tensors is to reduce the range of the representation. This relies on the assumption that maintaining small differences between tensor values that are close together is more important than representing the absolute extreme values or the outliers. Ideally, the numerical distribution of values in the mapped INT8 tensor representations should be as close as possible to the distribution of values for FP32 tensors. One way to measure this “closeness“ is to use the Kullback-Leibler Divergence (KL-Divergence) (Kullback & Leibler, 1951) metric between the histograms of FP32 and INT8 tensors. By iteratively choosing different Min and Max threshold values and mapping them to their respective extrema in the INT8 representation, we are able to find optimal Min and Max values that minimize the KL divergence between the INT8 and FP32 tensors. We refer to this process as calibration within the quantization workflow. This idea was first introduced in (Migacz, 2017).

We chose 600 random length samples out of 3003 sentences in the validation dataset as calibration data for quantization. The MatMul input tensors in Transformer LT graph come from three types of distributions, as classified from inspection of histogram values from inferring on the calibration dataset shown in Figure 2. The sparse tensors, when quantized, result in unacceptable accuracy degradation, and are kept unquantized i.e. with FP32 data type. For the other two types of tensors, they can be thresholded to get a reasonable accuracy degradation. The inputs to the MatMul in this model are both signed FP32, as opposed to the expected case of one signed weight and an unsigned activation. Thus, there is a need to quantize one of the tensors to unsigned INT8, which is described in detail in subsection 5.2. The way to generate the thresholds for this conversion of a signed FP32 tensor to unsigned INT8 tensor affects the overall accuracy of the model. We determined the positive and negative thresholds using three separate ways, referred to as quantization modes:

Table 1. Effects of calibration modes on the accuracy

Mode	BLEU score	Drop in Accuracy
Naïve quantization	NA	NA
Symmetric	27.30	0.38
Independent	27.33	0.35
Conjugate	27.26	0.421

1. Symmetric calculates the KL-divergence on the entire distribution. Here,

$$Threshold_{Min} = -Threshold_{Max}$$

2. *Independent* separates the distribution about value zero and calculates $Threshold_{Min}$ and $Threshold_{Max}$ independently

3. *Conjugate* separates the distribution about zero and calculates thresholds independently, but reports

$$\begin{cases} Threshold_{Max} = \max(|Max|, |Min|) \\ Threshold_{Min} = -Threshold_{Max} \end{cases}$$

Table 1 shows the effect of different quantization modes on the final acuracy (BLEU score). One test of the need for early thresholding using KL-divergence is to perform naïve quantization on all the MatMuls that do not have a sparse input. The graph generated by this method could not emit a STOP token during inference, giving out garbage translations. The BLEU score is unavailable, marked as NA in the table. This proves the need to use early thresholding. It can also be observed that independently calculating the thresholds for positive and negative halves of the histogram results in the least drop in accuracy. In this case, the min and max thresholds might not be symmetric about zero, causing the quantized tensors to have a non-zero value for the offset. This results in the QuantizedMatMul kernel being slightly slower than the case where the offsets are zero. Thus, we use the symmetric mode for threshold calculation at a small cost of 0.03% in accuracy. Note that we ended up not quantizing the tensors with Sparse histogram (appearing in 12 out of 97 MatMuls). Tensors with narrow and Gaussian distributions are quantized.

5. Improving Performance

Objective behind quantization is to leverage VNNI instructions and explore opportunities to further improve inference performance on Intel® Architecture(IA). In the process, we reduced total number of operations in the graph by removing redundant operations and reordering operations maintaining correctness. We introduced new optimizations in the MKL-DNN kernel implementations, found a new way of ordering input sentences and parallelized execution of multiple batches. These techniques are described in the following sections.

5.1. Experimental Setup

FP32 and INT8 performance are evaluated on 2S Intel ® Xeon ® Platinum 8168(24 cores per socket) processors and Intel ® Xeon ® Platinum 8268(24 cores per socket) processors, respectively. Both are tested with TensorFlow 1.12.0 built with VNNI, Intel ® MKL mklml_lnx_2019.0.3.20190119, Python 2.7 on CentOS 7.5.

5.2. Optimizing Quantized MatMuls

As seen in the operation timing splits in Figure 7, MatMuls take 40% of time, and are the first target for quantization. Quantized MatMul kernel in Intel® MKL BLAS accepts one signed and one unsigned INT8 matrix and accumulates results into a signed INT32 tensor. TensorFlow uses the open-source GEMMLOWP library for this integer matrix-matrix multiplication (Team, 2016). However, the GEMMLOWP MatMul kernels are not IA optimized. We integrated Intel ® MKL integer GEMM APIs to TensorFlow CPU backend. The performance of MKL MatMul in FP32 and INT8 configurations are shown in Figure 3. Figure 3a shows INT8 MatMuls using VNNI provides a speed-up of 3.7X over FP32 MatMuls using AVX512. Similarly, the speed-up provided by VNNI over AVX512 for INT8 MatMul is 2.3X.

We further profiled the workload to capture the matrix dimensions from the Transformer model and compared the INT8 and FP32 performance of these matrix sizes. Figure 3b shows that on average INT8 MatMuls achieve 2.4X speedup over FP32 MatMuls.

5.3. Optimizing GatherNd

The GatherNd operation on N-dimensional tensors uses the input indices to perform a Gather on the input tensor to form the output tensor. In total, there are 40 such GatherNd operations in the Transformer model which occur in the decoder while loop. These operations involve copy on large tensors, making it one of the most expensive operations due to beam search. There is no obvious compute benefit for quantizing GatherND. Looking into the GatherNd implementation in TensorFlow, we found that memory copy time dominates the operation time. Hence, we reduced amount of data copied by changing the data type of the tensors to INT8 via quantizing GatherNd.

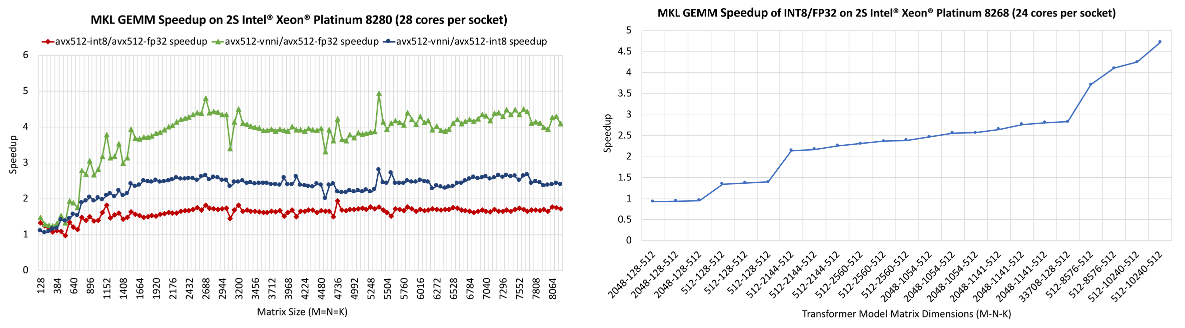
In the Transformer computational graph, the decoder while loop contains GatherNd operations as shown in Figure 4. The naïve way of quantizing GatherNd involves adding a Quantize and Dequantize node before and after the GatherNd node. This, however adds an overhead, which reduces the overall speedup. We however managed to minimize the extra cost of this Dequantize by repositioning the existing the Quantizes and Dequantizes in the graph due to QuantizedMatMul. With these changes we reduced the copy size by 3.8X for the validation dataset. The execution time of the GatherNd operation alone was reduced by 5X.

5.4. Sorting Input Sentences

In machine translation, the inputs to the network have varying sequence lengths. When input sentences are batched together, all the sentences except the longest sentence in the batch are padded to the sequence length of the longest sentence in each batch. This adds an overhead per batch in terms of wasted computations for the pad tokens. To work around this issue, it is important to sort the input validation dataset to keep the padding added to each batch to a minimum. There are different sorting methodologies such as sorting based on the number of words in each input sentence or token sorting based on the number of tokens in each input sentences. We have found that inference performance with sorting based on the number of tokens gives us an improvement of 28% over inference performance with sorting based on the word count of the input sentence.

5.5. Eliminating Operations from Graph

As discussed in subsection 4.2, finding thresholds using KL-divergence method eliminated the need for computing absolute Min and Max of tensors in the graph. These threshold values are inserted as Const operations in the graph. This further removed some of the Reshape operations. We also eliminated Requantize and RequantizationRange operations for tensors which were feeding in to unquantized operations. We used a Dequantize operation to convert INT32 to FP32 directly as shown in Figure 5. These removals contributed to reducing the total number of operations in the quantized compute graph. Additional quantize/dequantize operations were also removed in the GatherNd quantization.



(a) (b)

Figure 3. Comparison of speedups of MKL GEMM INT8 vs FP32 for different matrix shapes

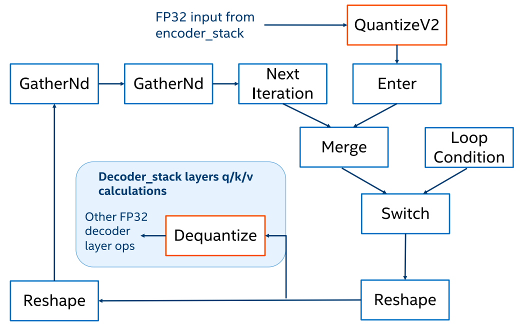

FP32 input from encoder_stack
QuantizeV2
GatherNd
GatherNd
Next Iteration
Enter
Merge
Loop Condition
Decoder_stack layers q/k/v calculations
Other FP32 decoder layer ops
Dequantize
Switch
Reshape
Reshape

Figure 4. Operations in the while loop of the Transformer model

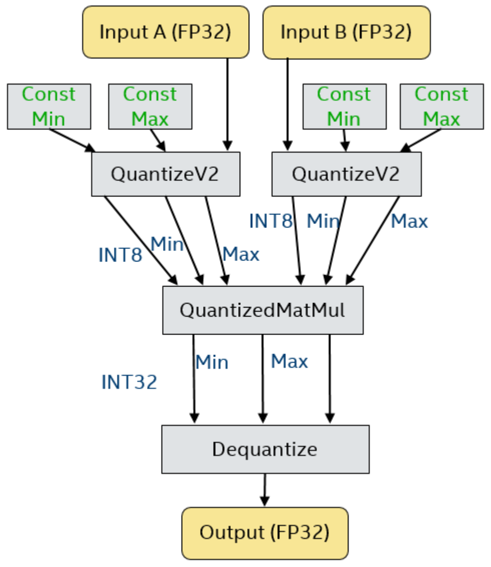


Figure 5. Schematic of Optimized Quantization for MatMuls in TensorFlow

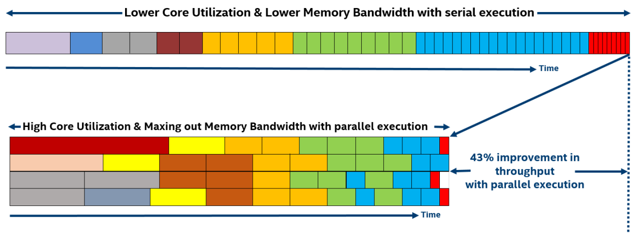

Lower Core Utilization & Lower Memory Bandwidth with serial execution
Time
High Core Utilization & Maxing out Memory Bandwidth with parallel execution
43% improvement in throughput with parallel execution
Time

Figure 6. Comparison of the serial and parallel execution techniques

5.6. Parallel Batching

The execution time of inference varies depending on the length of the sentences in the batch. This due to there being more number of operations such as matmuls with increased sentence length and due to increased decode steps. We have observed that there is a significant difference in the CPU core utilization depending on the sentence size. Batches of longer sentences make more efficient use of CPU cores while batches of shorter sentences fail to do so. Hence serially executing these batches is inefficient. We hypothesized that one way to increase CPU utilization would be to pack one or more batches of longer sentences with batches of shorter sentence.

The optimized methodology we follow is that we first create a parent TensorFlow session which in turn creates a batch queue. The parent process also creates children processes which are affinitized to specific subset of CPU cores and also affinitized to their local memory node using core and NUMA affinity settings. The children processes pick batches of input sentences from the batch queue and perform inference. The input sentences are ordered by decreasing token count before being added to the batch queue.

Note that the worker processes dequeue batches asynchronously from the batch queue and perform inference. Hence, as multiple batches of long and short

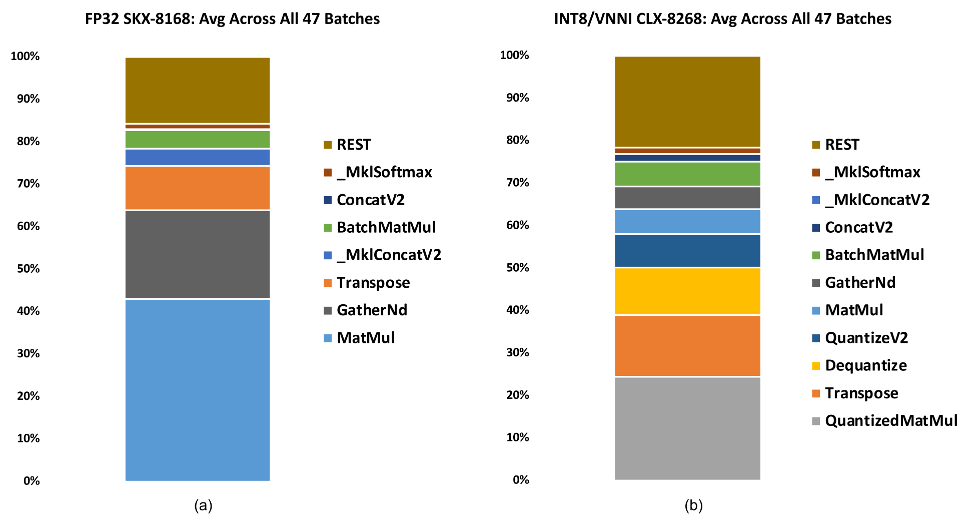


Figure 7. Distribution of percentage operation times in FP32 vs INT8 graph

sentences are processed in parallel, the CPU utilization significantly improves. As a result of this, we also observe a 43% improvement in throughput as shown in Figure 6. We use multiple inference streams per node as described in (Saletore et al., 2018).

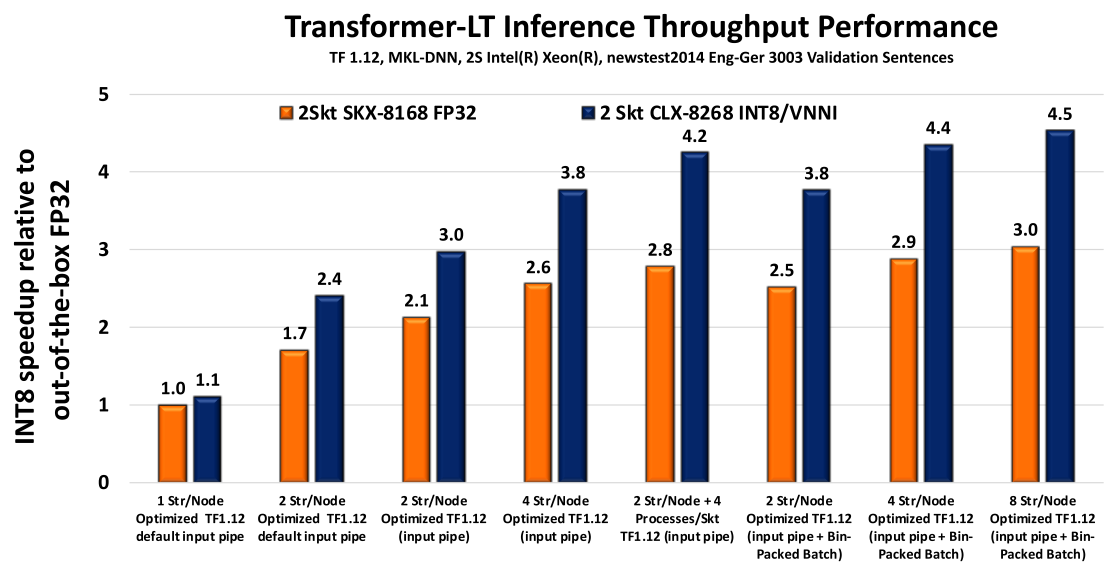
6. Throughput Performance Results

The inference performance of Transformer model for both INT8 and FP32 graphs are evaluated with newstest2014 dataset. A mini batch size of 64 is used in all experiments. In Figure 7, percentage of operation times are shown in different colors. MatMul is the major operation that accounts for 43% of the FP32 execution time on Intel® Xeon® 8168 processor. In case of INT8/VNNI quantized graph, some of these MatMuls are replaced with QuantizedMatMuls reducing the percentage of time spent in matrix multiplications. However, the quantization of MatMuls resulted in overheads such as Dequantize and QuantizeV2 in the INT8 graph. GatherND, another operation that took up a significant portion of the FP32 computation, also significantly reduced its INT8 percentage through the optimization described in subsection 5.3.

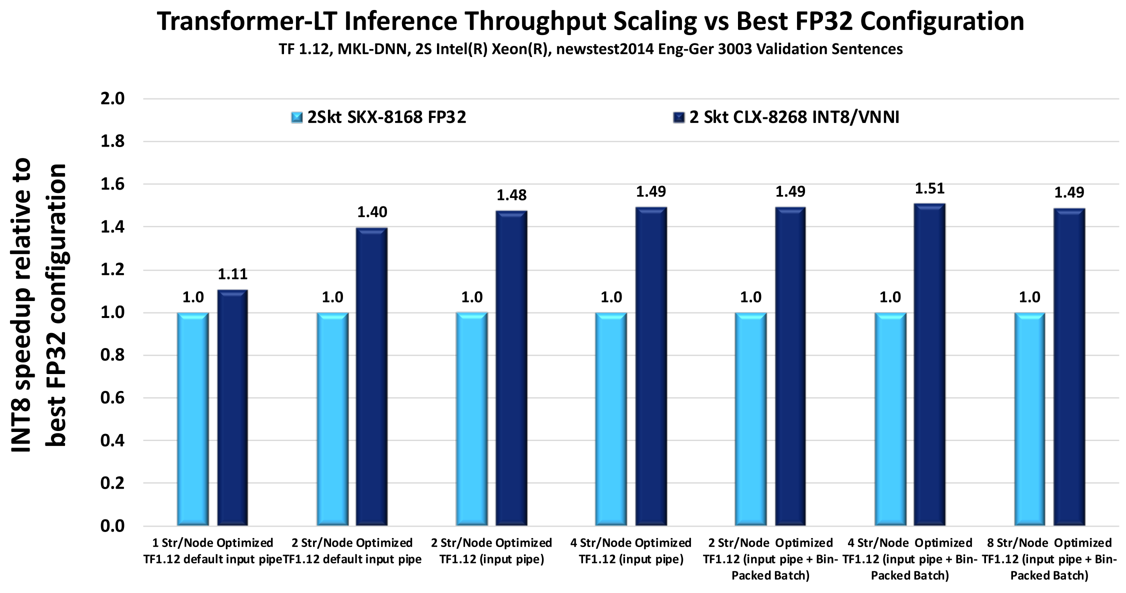
Figure 8 compares the overall inference throughput of the Transformer model with INT8/VNNI optimizations on 2S Xeon® 8268 with optimized FP32 on 2S Xeon® 8168 platforms. The first two bars in Figure 8 are throughput results obtained by using default word-sorted input data measured with 1 and 2 streams per node respectively. The next two sets use token-sorted input data on 2 and 4 streams per node respectively. The last three sets use both token-sorted and parallel batching with 2, 4, and 8 streams per node respectively. Figure 8a shows that we were able to achieve up to 4.5X throughput performance scaling with INT8/VNNI quantization relative to the out-of-the-box FP32 performance with all of our optimizations. However, our input pipeline and system level optimizations effectively improved FP32 performance by 3X. Figure 8b shows throughput performance scaling using INT8/VNNI relative to best FP32 SKX-8168 with the best system configuration. The highest INT8 throughput is achieved with 2 inference streams/socket with token sorting and parallel batching resulting in a scaling of 1.51X.

7. Conclusion

In this work we have quantized the Transformer machine language translation model in TensorFlow and maintained less than 0.5% drop in accuracy. Such a model, that has operations like Softmax and Layer Normalization in between quantizable operations like MatMul can still be selectively quantized to get a speed-up over FP32. Along with leveraging the VNNI instructions to get a speedup on the operation itself, other techniques can be used to get additional performance benefits. We optimized the compute graph by reducing number of operations, improved kernels of key operations such as MatMuls and GatherNd, optimized order of sentences in the input pipeline and finally used parallel batching to achieve the throughput gains of 1.5X.



(a)



(b)

Figure 8. Throughput Performance of INT8/VNNI on 2S Xeon® 8268 vs FP32 2S Xeon® 8168

Acknowledgements

We acknowledge the contributions from the Intel® TensorFlow Direct Optimization (TFDO) team and the Intel® Math Kernel library (MKL) team.

References

Alom, M. Z., Moody, A. T., Maruyama, N., Essen, B. C. V., and Taha, T. M. Effective quantization approaches for recurrent neural networks. *CoRR*, abs/1802.02615, 2018. URL http://arxiv.org/abs/1802.02615.

AWS. How amazon translate works, 2018. URL https://docs.aws.amazon.com/translate/latest/dg/how-it-works.html.

Chiu, C., Sainath, T. N., Wu, Y., Prabhavalkar, R., Nguyen, P., Chen, Z., Kannan, A., Weiss, R. J., Rao, K., Gonina, K., Jaitly, N., Li, B., Chorowski, J., and Bacchiani, M. State-of-the-art speech recognition with sequence-to-sequence models. *CoRR*, abs/1712.01769, 2017. URL http://arxiv.org/abs/1712.01769.

Courbariaux, M. and Bengio, Y. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016. URL http://arxiv.org/abs/1602.02830.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

Fomenko, E. Understanding new vector neural network instructions (vnni), 2018. URL https://aidc.gallery.video/detail/video/5790616836001/understanding-new-vector-neural-network-instructions-vnni.

Google, A. . M. L. P. Translating text, 2018. URL https://cloud.google.com/translate/docs/translating-text.

Guu, K., Hashimoto, T. B., Oren, Y., and Liang, P. Generating sentences by editing prototypes. *CoRR*, abs/1709.08878, 2017. URL http://arxiv.org/abs/1709.08878.

Gysel, P., Pimentel, J., Motamedi, M., and Ghiasi, S. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, PP:1–6, 03 2018. doi: 10.1109/TNNLS.2018.2808319.

Hou, L. and Kwok, J. T. Loss-aware weight quantization of deep networks. *CoRR*, abs/1802.08635, 2018. URL http://arxiv.org/abs/1802.08635.

Hu, M., Peng, Y., and Qiu, X. Mnemonic reader for machine comprehension. *CoRR*, abs/1705.02798, 2017. URL http://arxiv.org/abs/1705.02798.

Jiaxiang Wu, Cong Leng, Y. W. Q. H. and Cheng, J. Quantized convolutional neural networks for mobile devices. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

Kullback, S. and Leibler, R. A. On information and sufficiency. *The Annals of Mathematical Statistics*, 22 (1):79–86, 1951. ISSN 00034851. URL http://www.jstor.org/stable/2236703.

Lei Ba, J., Kiros, J. R., and Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

LingoTek. Api 5 basic integration, 2019. URL https://devzone.lingotek.com/basic-integration-5.

Mellempudi, N., Kundu, A., Mudigere, D., Das, D., Kaul, B., and Dubey, P. Ternary neural networks with fine-grained quantization, 2017. URL https://arxiv.org/pdf/1705.01462.pdf.

Microsoft, C. S. Machine translation, 2018. URL https://www.microsoft.com/en-us/translator/business/machine-translation/.

Migacz, S. 8-bit inference with tensorrt, 2017. URL http://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf.

Park, E., Yoo, S., and Vajda, P. Value-aware quantization for training and inference of neural networks. In *The European Conference on Computer Vision (ECCV)*, September 2018.

Saletore, V., Karkada, D., and Datta, K. Boosting deep learning training and inference performance on intel®xeon®and intel®xeon phi$^{\text{TM}}$processors, 2018. URL https://software.intel.com/en-us/articles/boosting-deep-learning-training-inference-performance-on-xeon-and-xeon-phi.

Team, G. G. Overview of gemmlowp design, 2016. URL https://github.com/google/gemmlowp/blob/master/doc/design.md.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. June 2017.

Wu, K. Tensorflow transformer official model, 2018. URL https://github.com/tensorflow/models/blob/master/official/transformer/test_data/newstest2014.en.

Wu, S., Li, G., Chen, F., and Shi, L. Training and inference with integers in deep neural networks. In *International Conference on Learning Representations ICLR*, 2018. URL https://arxiv.org/pdf/1802.04680.pdf.

Notices and Disclaimers

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps. Intel technologies features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer. No computer system can be absolutely secure. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

FTC Optimization Notice: Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice Revision #20110804 Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks. Cost reduction scenarios described are intended as examples of how a given Intel-based product, in the specified circumstances and configurations, may affect future costs and provide cost savings. Circumstances will vary. Intel does not guarantee any costs or cost reduction. The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document. Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate. Intel, the Intel logo, Atom, Xeon and others are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others. © 2019 Intel Corporation.