



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

DEPARTAMENTO DE INFORMÁTICA
DE SISTEMAS Y COMPUTADORES

Efficient Mixed-Precision Inference for Vision Transformers

Piotr Kluska

Advisors:

Prof. Enrique Salvador Quintana Ortí
Dr. Florian Scheidegger
Dr. A. Cristiano I. Malossi

December 2024

Acknowledgements

First, I would like to express my immense appreciation to my supervisors, Prof. Enrique Salvador Quintana-Ortí, Dr. Florian Scheidegger, and Dr. Cristiano Malossi. Our discussions, guided by your advice, were a cornerstone of this thesis. Thank you for keeping me on track and allowing me to pursue the deepest insights into the quantization of deep learning models. I am humbled to have had three exceptional advisors during my Ph.D. I am grateful for Enrique’s reviews of my papers and presentations. As well as every discussion, starting from the beginning, recommendations, and keeping me up to the highest standards until the very end. I would like to thank Florian for every idea we discussed, showing me that being concise but detailed is appreciated by the readers of our publications. Finally, I am grateful to Cristiano for the freedom to pursue a research topic and to seek the practical applicability of our research.

I would like to thank my PhD colleagues at IBM Research Zürich and Universitat Politècnica de València for sharing a journey with me: Thomas Frick, Brown Ebouky, Niccolo Avogaro, and Jie Lei. I thank my co-author, Adrian Castelló, for your collaboration and dedication and Andrea Bartezzaghi for productive technical discussions.

To Dominika Kwiatkowska for being a light even during the darkest hours. Thank you for always having my back, even when we are apart.

To all fellow MSCA PhD students and supervisors in the Approximate Computing for Power and Energy Optimisation project. It was a pleasure to walk with you along the PhD path. I hope that one day, our paths cross again.

Lastly, I would like to acknowledge gratefully funding from European Union’s Horizon 2020 Research and Innovation Programme under the Marie Skłodowska Curie grant agreement No. 956090 (APROPOS: Approximate Computing for Power and Energy Optimisation).

Abstract

Recent advances in deep learning (DL) have been achieved by scaling the number of model parameters and training data. The Transformer is the most widely used DL model architecture in natural language processing. Its key feature is the attention mechanism, which allows the model to focus dynamically on the relevant context in the global latent space. This has led to scaling models in natural language processing with up to 405 billion parameters, equivalent to 810 gigabytes (GB) of memory in 16-bit floating point (FP16). At this level of accuracy, a model of this size requires several accelerator compute nodes. The Transformer architecture has been successfully transferred to the computer vision domain as the Vision Transformer (ViT). Similarly, scaling the number of parameters in the ViT architecture increased the model's predictive power. In addition, modifications to the ViT architecture led to new architectures such as the Data efficient Image Transformer (DeiT), Swin Transformer, and DeiT3. Each of these improved upon the shortcomings of the original architecture. Nevertheless, these models require substantial computing power and energy to process the images at scale.

The emergence of Artificial Intelligence (AI) systems and applications will require DL models to run efficiently with a conscious energy consumption, as it is expected that, by 2040, the energy consumed by devices will exceed our energy production capability. To tackle this, we investigate the compression methods for ViT architectures to reduce the model's required size and translate the computation to more energy-efficient data types. The methods and algorithms presented in this dissertation allow the ViT architectures to operate with less energy consumed and reduced latency at a similar level of predictive performance to the reference model.

Firstly, we comprehensively evaluate the effect of the post-training quantization on the ViT, DeiT, Swin Transformer, and DeiT3 models. We show that ViT and DeiT3, after quantization, lose their predictive power, while DeiT and Swin do not. We hypothesize that the regularization applied during training positively

affects the quantization’s robustness. Next, we perform per-layer analysis using the signal-to-quantization-noise ratio (SQNR), which measures the latent signal going through a quantized network compared to the reference FP32 DL model. We show that a correlation exists between the SQNR value and quantization error. Moreover, we propose an easy yet effective post-training quantization method that utilizes mixed-precision computation, allowing us to compress the models up to 90%. As a result, our models approach the fully quantized DL models in size while keeping the predictive performance close to the FP32.

Next, we propose a novel post-training quantization method - Hybrid Quantization (HQ). HQ uses the property of the ViTs, which are mainly composed of linear layers. As a result, we design an automatic algorithm that selects the linear layer for static or dynamic quantization based on the SQNR metric. Our HQ method improves the predictive power performance compared to the static quantization in 12/12 ViT, 3/6 DeiT, 6/6 DeiT3, and 6/6 Swin Transformer models on the ImageNet1K validation dataset. Furthermore, we evaluate the latency of HQ models on three hardware environments: an Intel Xeon 5218 Gold CPU, a mobile Apple A15 Bionic CPU, and an NVIDIA A100 GPU. We observe up to 1.15, 1.28, and 1.68 average speedups compared to the dynamic quantization for ViT models, respectively.

Lastly, we design and implement a mixed-precision attention mechanism in the Triton language. Our mixed-precision attention mixes an 8-bit integer (INT8) and FP16 computation to achieve higher throughput and numerical stability than the reference implementation of FlashAttention in Triton. We show that using a domain-specific language with a compiler can match heavily specialized GPU kernels. Moreover, we open-source our QAttn framework. In this library, we focus on the integration of the PyTorch post-training quantization ecosystem. We extend the PyTorch quantization with custom kernels for quantized matrix multiplication and our mixed-precision attention. We show that our kernels improve the throughput of the ViT model by up to 7.34 compared to the FP32 reference model. Moreover, our framework generalizes to newer foundation models like the Segment Anything Model (SAM). We achieve over 5x more images processed per second for the base and large variants without mean intersection over union (mIOU) drop over the COCO2017 validation set.

In summary, in this thesis, we holistically address the problem of post-training quantization of ViT models. We propose a novel method to tackle the quantization of ViT architecture. In addition, we open-source the QAttn framework, which implements quantized GPU kernels in Triton and integrates with the PyTorch framework. In our research, we demonstrate memory and latency reduction compared to the reference DL model. Moreover, our work lays the foundation for further research into the compression of ViT models.

Resumen

Los avances recientes en el aprendizaje profundo (DL) se han logrado escalando el número de parámetros del modelo y los datos de entrenamiento. El Transformer es la arquitectura de modelo de DL más utilizada en el procesamiento del lenguaje natural. Su característica clave es el mecanismo de atención, que permite al modelo centrarse dinámicamente en el contexto relevante en el espacio latente global. Esto ha permitido escalar modelos en el procesamiento del lenguaje natural con hasta 405.000 millones de parámetros, lo que equivale a 810 gigabytes (GB) de memoria en coma flotante de 16 bits (FP16). A este nivel de precisión, un modelo de este tamaño requiere varios nodos de cálculo acelerador. La arquitectura Transformer se ha trasladado con éxito al ámbito de la visión por ordenador como Vision Transformer (ViT). Del mismo modo, la ampliación del número de parámetros de la arquitectura ViT aumentó la capacidad de predicción del modelo. Además, las modificaciones de la arquitectura ViT dieron lugar a nuevas arquitecturas como el Transformador de Imagen Eficiente en Datos (DeiT), el Transformador Swin y el DeiT3. Cada una de ellas mejoraba las deficiencias de la arquitectura original. No obstante, estos modelos requieren una potencia de cálculo y una energía considerables para procesar las imágenes a escala.

La aparición de sistemas y aplicaciones de Inteligencia Artificial (IA) exigirá que los modelos de DL funcionen de forma eficiente con un consumo de energía consciente, ya que se prevé que, para 2040, la energía consumida por los dispositivos superará nuestra capacidad de producción energética. Para hacer frente a esto, investigamos los métodos de compresión para arquitecturas ViT con el fin de reducir el tamaño requerido del modelo y trasladar el cómputo a tipos de datos más eficientes energéticamente. Los métodos y algoritmos presentados en esta tesis permiten a las arquitecturas ViT funcionar con un menor consumo energético y una latencia reducida con un nivel de rendimiento predictivo similar al del modelo de referencia.

En primer lugar, evaluamos exhaustivamente el efecto de la cuantización posterior al entrenamiento en los modelos ViT, DeiT, Swin Transformer y DeiT3. Demostramos que ViT y DeiT3, tras la cuantización, pierden su poder predictivo, mientras que DeiT y Swin no. Nuestra hipótesis es que la regularización aplicada durante el entrenamiento afecta positivamente a la robustez de la cuantización. A continuación, realizamos un análisis por capas utilizando la relación señal/ruido de cuantización (SQNR), que mide la señal latente que atraviesa una red cuantizada en comparación con el modelo FP32 DL de referencia. Demostramos que existe una correlación entre el valor de SQNR y el error de cuantización. Además, proponemos un método de cuantización post-entrenamiento sencillo pero eficaz que utiliza el cálculo de precisión mixta, lo que nos permite comprimir los modelos hasta un 90%. Como resultado, nuestros modelos se aproximan en tamaño a los modelos DL totalmente cuantizados, al tiempo que mantienen el rendimiento predictivo cercano al FP32.

A continuación, proponemos un nuevo método de cuantización post-entrenamiento: la Cuantización Híbrida (HQ). HQ utiliza la propiedad de los ViTs, que se componen principalmente de capas lineales. Como resultado, diseñamos un algoritmo automático que selecciona la capa lineal para la cuantización estática o dinámica basándose en la métrica SQNR. Nuestro método HQ mejora el rendimiento del poder predictivo en comparación con la cuantización estática en los modelos 12/12 ViT, 3/6 DeiT, 6/6 DeiT3 y 6/6 Swin Transformer en el conjunto de datos de validación ImageNet1K. Además, evaluamos la latencia de los modelos HQ en tres entornos de hardware: una CPU Intel Xeon 5218 Gold, una CPU móvil Apple A15 Bionic y una GPU NVIDIA A100. Observamos una aceleración media de hasta 1,15, 1,28 y 1,68 en comparación con la cuantización dinámica de los modelos ViT, respectivamente.

Por último, diseñamos e implementamos un mecanismo de atención de precisión mixta en el lenguaje Triton. Nuestra atención de precisión mixta mezcla un entero de 8 bits (INT8) y un cálculo FP16 para lograr un mayor rendimiento y estabilidad numérica que la implementación de referencia de FlashAttention en Triton. Demostramos que el uso de un lenguaje específico del dominio con un compilador puede adaptarse a núcleos de GPU muy especializados. Además, desarrollamos nuestro marco QAttn en código abierto. En esta biblioteca, nos centramos en la integración del ecosistema de cuantización post-entrenamiento de PyTorch. Ampliamos la cuantización de PyTorch con kernels personalizados para la multiplicación de matrices cuantizadas y nuestra atención de precisión mixta. Demostramos que nuestros kernels mejoran el rendimiento del modelo ViT hasta en un 7,34 en comparación con el modelo de referencia FP32. Además, nuestro marco de trabajo se generaliza a modelos de base más recientes, como el Segment Anything Model (SAM). Conseguimos más de 5 veces más imágenes procesadas

por segundo para la variante base y la variante grande sin caída de la intersección media sobre la unión (mIOU) en el conjunto de validación COCO2017.

En resumen, en esta tesis abordamos de forma holística el problema de la cuantización post-entrenamiento de los modelos ViT. Proponemos un método novedoso para abordar la cuantización de la arquitectura ViT. Además, desarrollamos en código abierto el framework QAttn, que implementa kernels cuantificados de GPU en Triton y se integra con el framework PyTorch. En nuestra investigación, demostramos la reducción de memoria y latencia en comparación con el modelo DL de referencia. Además, nuestro trabajo sienta las bases para futuras investigaciones sobre la compresión de modelos ViT.

Resums

Els avanços recents en aprenentatge profund (DL) s'han aconseguit ampliant el nombre de paràmetres del model i les dades d'entrenament. El Transformer és l'arquitectura de model DL més utilitzada en el processament del llenguatge natural. La seua característica clau és el mecanisme d'atenció, que permet al model enfocar-se dinàmicament en el context rellevant en l'espai latent global. Això ha portat a l'ampliació dels models en el processament del llenguatge natural amb fins a 405 mil milions de paràmetres, equivalents a 810 gigabytes (GB) de memòria en punt flotant de 16 bits (FP16). A aquest nivell de precisió, un model d'aquesta mida requereix diversos nodes d'acceleradors de càlcul. L'arquitectura Transformer s'ha transferit amb èxit al domini de la visió per computador com el Vision Transformer (ViT). De la mateixa manera, l'ampliació del nombre de paràmetres en l'arquitectura ViT va augmentar el poder predictiu del model. A més, les modificacions a l'arquitectura ViT van donar lloc a noves arquitectures com el Data efficient Image Transformer (DeiT), Swin Transformer, i DeiT3. Cadascun d'aquests va millorar les mancances de l'arquitectura original. No obstant això, aquests models requereixen una gran potència de càlcul i energia per processar les imatges a gran escala.

L'aparició de sistemes i aplicacions d'Intel·ligència Artificial (IA) requerirà que els models d'Aprenentatge Profund (DL) funcionen de manera eficient amb un consum energètic conscient, ja que s'espera que, per a l'any 2040, l'energia consumida pels dispositius superarà la nostra capacitat de producció energètica. Per a abordar això, investiguem els mètodes de compressió per a les arquitectures ViT per a reduir la mida requerida pel model i traduir el càlcul a tipus de dades més eficients en energia. Els mètodes i algorismes presentats en aquesta dissertació permeten que les arquitectures ViT operen amb menys consum d'energia i una latència reduïda, mantenint un nivell de rendiment predictiu similar al model de referència.

En primer lloc, avaluem de manera exhaustiva l'efecte de la quantificació

posterior a l'entrenament en els models ViT, DeiT, Swin Transformer i DeiT3. Mostrem que ViT i DeiT3, després de la quantificació, perden el seu poder predictiu, mentre que DeiT i Swin no ho fan. Hipotetitzem que la regularització aplicada durant l'entrenament afecta positivament la robustesa de la quantificació. A continuació, realitzem una anàlisi capa per capa utilitzant la ràtio de senyal a soroll de quantificació (SQNR), que mesura el senyal latent que travessa una xarxa quantificada en comparació amb el model de DL FP32 de referència. Demostrem que existeix una correlació entre el valor de SQNR i l'error de quantificació. A més, proposem un mètode de quantificació posterior a l'entrenament senzill però efectiu que utilitza càlcul de precisió mixta, la qual cosa ens permet comprimir els models fins a un 90%. Com a resultat, els nostres models s'acosten a la mida dels models de DL totalment quantificats, mantenint un rendiment predictiu proper al de FP32.

A continuació, proposem un nou mètode de quantificació posterior a l'entrenament: la Quantificació Híbrida (HQ). HQ utilitza la propietat dels ViT, que estan compostos principalment per capes lineals. Com a resultat, dissenyem un algorisme automàtic que selecciona la capa lineal per a quantificació estàtica o dinàmica basant-se en la mètrica SQNR. El nostre mètode HQ millora el rendiment del poder predictiu en comparació amb la quantificació estàtica en 12/12 models ViT, 3/6 DeiT, 6/6 DeiT3 i 6/6 models Swin Transformer en el conjunt de dades de validació ImageNet1K. A més, avaluem la latència dels models HQ en tres entorns de maquinari: una CPU Intel Xeon 5218 Gold, una CPU mòbil Apple A15 Bionic i una GPU NVIDIA A100. Observem millores de velocitat mitjana de fins a 1.15, 1.28 i 1.68 vegades, respectivament, en comparació amb la quantificació dinàmica per als models ViT.

Finalment, dissenyem i implementem un mecanisme d'atenció de precisió mixta en el llenguatge Triton. La nostra atenció de precisió mixta combina el càlcul amb enter de 8 bits (INT8) i FP16 per a aconseguir un major rendiment i estabilitat numèrica que la implementació de referència de FlashAttention en Triton. Demostrem que l'ús d'un llenguatge específic de domini amb un compilador pot igualar els nuclis GPU altament especialitzats. A més, publiquem en codi obert el nostre marc QAttn. En aquesta biblioteca, ens centrem en la integració de l'ecosistema de quantificació posterior a l'entrenament de PyTorch. Ampliem la quantificació de PyTorch amb nuclis personalitzats per a la multiplicació de matrius quantificades i la nostra atenció de precisió mixta. Mostrem que els nostres nuclis milloren el rendiment del model ViT fins a 7,34 vegades en comparació amb el model de referència FP32. A més, el nostre marc es generalitza a models fonamentals més nous com el Segment Anything Model (SAM). Aconsegüim processar més de 5 vegades més imatges per segon per a les variants base i gran sense pèrdua de la intersecció sobre la unió mitjana (mIOU) en el conjunt de validació COCO2017.

Contents

Acknowledgements	v
Abstract	vii
List of Figures	xix
List of Tables	xxv
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	2
1.3 Structure of the Thesis	3
1.4 List of publications	3
2 Foundations	5
2.1 Deep Learning	5
2.2 Computer Vision	7
2.2.1 Image Classification	8
2.2.2 Object Detection	9
2.2.3 Segmentation	10
2.2.4 Convolutional Neural Networks	12
2.3 Vision Transformers	13
2.4 Quantization	17
2.4.1 Static Quantization	21
2.4.2 Dynamic Quantization	22

2.5	Related Work	23
2.5.1	Knowledge distillation	23
2.5.2	Mixture of experts	24
2.5.3	Sparsification	24
2.5.4	Quantization	25
2.6	Deep learning frameworks and hardware	29
2.7	Conclusions	31
3	Challenges in Post-Training Quantization of Vision Transformers	33
3.1	Introduction	33
3.2	Background	34
3.3	Methodology	36
3.4	Results	37
3.4.1	Post-training Quantization of Vision Transformers	38
3.4.2	Activation Distribution and Sensitivity Analysis	40
3.4.3	Mixed-precision Quantization of Vision Transformers	47
3.5	Summary and Conclusions	52
4	Hybrid Quantization	55
4.1	Introduction	55
4.2	Related Work	56
4.3	Hybrid Quantization	57
4.4	Experimental Setup	58
4.5	Results	60
4.6	Summary and Conclusions	72
5	Efficient GPU Kernels for Mixed-Precision Vision Transformers	75
5.1	Introduction	75
5.2	Motivation	77
5.3	Triton	78
5.4	Quantized Matrix Multiplication	79
5.5	Mixed-Precision Attention	83
5.6	QAttn	87
5.7	Experimental Setup	89
5.8	Results	90
5.8.1	Matrix Multiplication Kernel	90
5.8.2	Attention Kernel	95
5.8.3	Vision Transformers	97
5.8.4	Segment Anything Model	99
5.9	Summary and Conclusions	101

6	Conclusions	103
6.1	Summary of Results	103
6.2	Outlook	104
	Acronyms	106
	Bibliography	111

List of Figures

1.1	Overview of thesis structure	4
2.1	Example deep neural network architecture. This network has two inputs with three input neurons. Followed by two hidden layers with five and three neurons, respectively. Lastly, the output layer consists of single neuron.	6
2.2	An example of an RGB (red, green, and blue) image with a cat. . . .	8
2.3	An overview of the image classification task. Given the input image, the DL model outputs the probability of an object in the image. . . .	10
2.4	An overview of the segmentation task. The model predicts the masks and labels of the objects in the image. Then, the masks can be used to visualize the predictions as an overlay on the image. The white pixels in the mask show the pixel belonging to the class.	12
2.5	Vision Transformer Model Overview: This diagram shows the architecture of the ViT, which takes an input image, divides the image into patches, and processes them through a series of transformer layers to produce an output representation that can be used for various computer vision tasks. Each Transformer layer consists of Layer Normalization, MHA, and MLP blocks with residual connections in between. The illustration of the Vision Transformer architecture was inspired by Dostovitskiy et al. [8]	14
2.6	This figure illustrates the MHA block within ViT encoder.	17

2.7	Predictive power (Top-1 Accuracy) vs. Number of Parameters for Image Classification DL Models on ImageNet1K [22] This figure shows the relationship between number of parameters and performance (Top-1 Accuracy) on ImageNet1K for several state-of-the-art image classification models, including Vision Transformers (ViT) [8], Data Efficient Image Transformers (DeiT [36] and DeiT3 [41]), Swin Transformers [40], ResNets [35], and ConvNeXt [27], demonstrating the trade-offs between model size and performance.	18
2.8	Static post-training quantization of DL model nodes. This figure shows an example of the node being quantized using static quantization. First, we insert the observers that compute the statistics of the input and output of the node. After calibration the observer nodes are removed and scaling factors and zero points (removed for brevity) are inserted into the function as a constant parameters.	21
3.1	This image displays attention heatmaps created by the final self-attention module of a ViT-B/16/224 model using sample ImageNet1K validation images. The color intensity corresponds to the softmax activation values, revealing the model's attention patterns and indicating the most important regions of the image for classification. Interestingly, some of the model's attention focuses on the background rather than the object itself.	35
3.2	This box plot shows the output activation distribution of the <i>blocks.2.mlp.fc1</i> fully connected layer in ViT, DeiT, and DeiT3. The input image size for each model is 224×224 , and the patch size is $P = 16$. The points outside the whiskers of the box plot represent outliers in the output activations.	41
3.3	This box plot shows the output activation distribution of the <i>blocks.2.mlp.fc2</i> fully connected layer in ViT, DeiT, and DeiT3. The input image size for each model is 224×224 , and the patch size is $P = 16$. The points outside the whiskers of the box plot represent outliers in the output activations.	42
3.4	This box plot shows the output activation distribution of the <i>blocks.10.attn.qkv</i> fully connected layer in ViT, DeiT, and DeiT3. The input image size for each model is 224×224 , and the patch size is $P = 16$. The points outside the whiskers of the box plot represent outliers in the output activations.	43

3.5	This box plot shows the output activation distribution of the <i>blocks.10.mlp.fc1</i> fully connected layer in ViT, DeiT, and DeiT3. The input image size for each model is 224×224 , and the patch size is $P = 16$. The points outside the whiskers of the box plot represent outliers in the output activations.	44
3.6	This box plot shows the output activation distribution of the <i>blocks.10.mlp.fc2</i> fully connected layer in ViT, DeiT, and DeiT3. The input image size for each model is 224×224 , and the patch size is $P = 16$. The points outside the whiskers of the box plot represent outliers in the output activations.	45
3.7	SQNR Ratio Analysis of ViT Models: This figure compares SQNR values for small and base ViT family models. SQNR of activations measured after each layer, illustrating the impact of quantization on intermediate representations.	46
3.8	SQNR Ratio Analysis of ViT Models: This figure compares SQNR values for small and base ViT family models. SQNR values between FP32 model weights and INT8 quantized weights.	46
3.9	Linear regression between average SQNR value of fully connected layers and quantization error of ViT, DeiT, and DeiT3. We observed a negative correlation between SQNR and quantization error with $\rho = -0.36$, 90% CI $[-0.67, 0.045]$. Circles indicate models belonging to one of the model families.	47
3.10	Effect of mixed-precision partial quantization on model memory size. Relative model size (x-axis) is shown for 50%, 75%, and 95% node quantization, with error bars indicating variation over ten runs. Baseline model sizes (FP32) are labeled on each bar.	51
4.1	A high-level overview of the HQ algorithms: during the tuning phase, the signal-to-noise ratio (SQNR) is calculated between the outputs of the models at the same linear node (f). The resulting model is a combination of static and dynamic quantized blocks used in the inference phase.	57
4.2	Percentage of dynamic quantized layers in the models: The X-axis displays the model, while the Y-axis represents the ratio of dynamic to static layers. Higher values indicate a higher percentage of dynamic layers in the model after HQ. The points on the graph represent the average ratio over five runs, with the standard deviation plotted. . . .	60

LIST OF FIGURES

4.3	Average model inference latency on the A15 Bionic CPU: each point represents a single quantization configuration of the selected HQ algorithm. We compared the latency of the proposed methods with INT8 and INT8-D. The measurements were averaged over 1000 samples. . .	61
4.4	Average inference latency of models on the A100 GPU-equipped workstation: each point represents a single quantization configuration of the selected HQ algorithm. We compared the latency of the proposed methods with INT8 and INT8-D. The measurements were averaged over 1000 samples.	62
4.5	CPU-only workstation: Average Intel Xeon 5218 Gold CPU inference latency of models. Each point represents a single quantization configuration. We report the average latency of linear layers over 1000 samples.	63
4.6	Latency vs. Accuracy Tradeoff of HQ Algorithms ViT-S/32/384 and DeiT3-L/16/224: This figure shows the tradeoff between latency and accuracy of the HQ algorithms compared to static (INT8) and dynamic (INT8-D) quantization. For each HQ variation, we plot the mean and standard deviation with a diamond, and we plot the best model out of five runs. Measurements were taken on an NVIDIA A100 GPU. The results above and to the left of the dashed red line indicate the improvement in accuracy over static quantization and latency over dynamic quantization, respectively.	64
4.7	Latency vs. Accuracy Tradeoff of HQ Algorithms ViT-B/32/384 and ViT-L/32/384: This figure shows the tradeoff between latency and accuracy of the HQ algorithms compared to static (INT8) and dynamic (INT8-D) quantization. For each HQ variation, we plot the mean and standard deviation with a diamond, and we plot the best model out of five runs. Measurements were taken on an NVIDIA A100 GPU. The results above and to the left of the dashed red line indicate the improvement in accuracy over static quantization and latency over dynamic quantization, respectively.	65
4.8	Latency vs. Accuracy Tradeoff of HQ Algorithms DeiT-S/16/224 and DeiT3-B/16/384: This figure shows the tradeoff between latency and accuracy of the HQ algorithms compared to static (INT8) and dynamic (INT8-D) quantization. For each HQ variation, we plot the mean and standard deviation with a diamond, and we plot the best model out of five runs. Measurements were taken on an NVIDIA A100 GPU. The results above and to the left of the dashed red line indicate the improvement in accuracy over static quantization and latency over dynamic quantization, respectively.	66

5.1	Triton autotune and heuristics decorators are used in the static matrix multiplication kernel. We provide various configurations on a number of stages and warps as well as block sizes that the autotuner tries to optimize over.	81
5.2	A simplified quantized static matrix multiplication kernel implemented in Triton. The matrix A is of size $M \times K$, matrix B is of size $K \times N$, while the output matrix C is of size $M \times N$. For brevity, we omit the bias parameter. Moreover, we for simplicity assume the K dimension to be even.	82
5.3	Mixed-precision attention kernel signature and autotuning options. This code block presents the kernel definition and initial phase of calculating the offsets and preparing the block pointers and accumulators for the kernel execution.	85
5.4	Mixed-precision attention kernel implementation. This code block presents the mixed-precision attention computation instructions. Here, we assume that the matrices might not be even in context length dimension. We check boundaries when we load and store the matrices to avoid wrong memory access.	86
5.5	An example of using the QAttn with torch.fx quantization workflow on the ViT-L/16/224 model.	87
5.6	An example of using the QAttn with torch.compile quantization workflow on the ViT-L/16/224 model.	88
5.7	Matrix multiplication with dimensions M, N and K over different batch sizes of ViT-S/32/224 linear layer. Reference Pytorch runs in FP32 and TensorRT in INT8. Our dynamic version accepts FP16 inputs with weights in INT8; our static implementations Triton and TVM run in INT8.	91
5.8	Matrix multiplication with dimensions M, N and K over different batch sizes of ViT-S/32/224 linear layer. Reference Pytorch runs in FP32 and TensorRT in INT8. Our dynamic version accepts FP16 inputs with weights in INT8; our static implementations Triton and TVM run in INT8.	92
5.9	Matrix multiplication with dimensions M, N and K over different batch sizes of ViT-B/16/224 linear layer. Reference Pytorch runs in FP32 and TensorRT in INT8. Our dynamic version accepts FP16 inputs with weights in INT8; our static implementations Triton and TVM run in INT8.	93

5.10	Matrix multiplication with dimensions M , N and K over different batch sizes of ViT-L/32/224 linear layer. Reference Pytorch runs in FP32 and TensorRT in INT8. Our dynamic version accepts FP16 inputs with weights in INT8; our static implementations Triton and TVM run in INT8.	94
5.11	Matrix multiplication with dimensions M , N and K over different batch sizes of ViT-L/32/384 linear layer. Reference Pytorch runs in FP32 and TensorRT in INT8. Our dynamic version accepts FP16 inputs with weights in INT8; our static implementations Triton and TVM run in INT8.	94
5.12	Performance comparison of various attention kernels with $H = 12$, $d = 64$, and $b = 512$: PyTorch scaled dot product attention, FP16 Triton reference implementation, dynamic mixed-precision attention, static quantization mixed-precision attention, and FlashAttention2. . .	96
5.13	Performance comparison of various attention kernels with $H = 16$, $d = 64$, and $b = 512$: PyTorch scaled dot product attention, FP16 Triton reference implementation, dynamic mixed-precision attention, static quantization mixed-precision attention, and FlashAttention2. . .	97
5.14	Relationship between predictive performance and image processing speed (images per second) for each SAM model, based on data type. Larger models show improved predictive power at the expense of reduced throughput. The blue line represents the reference FP32 SAM implementation, the orange line represents the Segment Anything Fast INT8-D*, while green and red lines are our dynamically and statically quantized kernels.	100

List of Tables

2.1	This table summarizes the computer vision tasks: image classification, object detection, semantic segmentation, instance segmentation, and panoptic segmentation. The table also shows the metrics typically used to evaluate the performance of these tasks.	9
2.2	Sequence length N (eq. (2.5)) relative to image size $H \times W$ and patch size P . This table shows the relationship of patch size and image size to sequence length. Each sequence length is incremented by one because the class token is prepended to the sequence length.	14
2.3	Architecture and parameter count of ViT models. This table provides an overview of the architecture and parameter counts for several ViT models, highlighting the variations in layer count, MLP size, hidden size, number of attention heads, and total parameters across models. .	15
2.4	Floating point datatype formats. This table lists the bit configurations for various floating-point data types, including the number of bits allocated to the sign bit, exponent, and mantissa, and illustrates the tradeoffs between precision and storage requirements.	19
2.5	Integer data type formats. This table lists the formats and value ranges of several integer data types, including the number of bits, minimum value, and maximum value for each type, highlighting the tradeoffs between precision and storage requirements.	20

3.1	Top-1 accuracy of post-training quantization on the ImageNet1K validation set for ViT and Swin Transformer models. We explored static (INT8) and dynamic quantization (INT8-D). The results are compared to FQ-ViT and PTQ4ViT. Model labels specify model name/patch size/input size. The best quantized top-1 score is highlighted in bold.	39
3.2	Top-1 accuracy of post-training quantization on the ImageNet1K validation set for DeiT and DeiT3 models. We report static (INT8) and dynamic quantization (INT8-D). The results are compared to FQ-ViT and PTQ4ViT. Model labels include model name/patch size/input size, and whether the model was distilled. The best quantized top-1 score is highlighted in bold.	40
3.3	Partial mixed-precision quantization results on ImageNet1K: This table presents the top-1 quantization error (mean $\pm$ standard deviation) for ViT, DeiT, and Swin tiny models on the ImageNet1K validation set. Results are averaged over ten runs. The lowest quantization error with the highest percentage of operations (Op.) quantized is highlighted in bold for each model.	48
3.4	Partial mixed-precision quantization results on ImageNet1K: This table presents the top-1 quantization error (mean $\pm$ standard deviation) for ViT, DeiT, DeiT3, and Swin small and base models on the ImageNet1K validation set. Results are averaged over ten runs. The lowest quantization error with the highest percentage of operations (Op.) quantized is highlighted in bold for each model.	49
3.5	Partial mixed-precision quantization results on ImageNet1K: This table presents the top-1 quantization error (mean $\pm$ standard deviation) for ViT and DeiT3 small and base models on the ImageNet1K validation set with input image size 384×384 in the top part. In the bottom part we present ViT, DeiT3, and Swin large and huge models. Results are averaged over ten runs. The lowest quantization error with the highest percentage of operations (Op.) quantized is highlighted in bold for each model.	50
4.1	Average top-1 accuracy with standard deviation on the ImageNet1K validation dataset computed over five runs. Our method improved the top-1 accuracy over the reference INT8 static model in 12/12 ViT, 3/6 DeiT, 6/6 DeiT3, and 4/4 Swin models.	68

4.2	Comparison between baseline static quantization (INT8), dynamic quantization (INT8-D), hybrid quantization (HQ1, HQ2, and HQ3), and FQ-ViT. We report top-1 accuracy on the ImageNet1K validation dataset. In this experiment we present the best model out of 5 runs. We reproduced and extended the results of FQ-ViT. In bold we mark the best result between HQ and FQ-ViT.	69
4.3	Average speedup latency improvement: this table presents the average speedup latency computed over five runs of the model compared to dynamic quantization. Each HQ configuration was executed on three hardware devices: iPhone 13 Pro with A15 CPU, baremetal with Intel Xeon 5218 Gold CPU, and a server with an NVIDIA A100 80GB GPU. The best speedup is shown in bold.	70
5.1	OPs by percentage and execution time for ViT-L/16/224. Number of layers, OPs, and latency are provided as a percentage [%] of the ViT model. We provide execution time for batch size $b = 1$ and $b = 128$. .	76
5.2	Comparison of top-1 accuracy on ImageNet1K validation dataset comparison between baselines and models with our kernels. We evaluated static (INT8) and dynamic (INT8-D) quantization of linear layers. INT8+A and INT8-D+A represent the addition of quantized mixed precision attention. Our runs are averaged over five repetitions. The results of PTQ4ViT and Zhang et al. are as reported by the authors. In bold, we mark the best static quantization result.	98
5.3	Throughout speedup (FP32 vs ours static) of ViT models measured over the ImageNet1K validation dataset.	99

Chapter 1

Introduction

According to the AI Index report by Stanford, in 2023 researchers and corporations released 149 Foundation Models (FM), which is twice as many as in 2022 [1]. Furthermore, in 2024, we have observed a growing number of FM models capable of processing multimodal data, including images, videos, and text [2, 3, 4]. The multimodal FM consists of a large language model (LLM) [5, 6, 7] and a Vision Transformer [8] that translates the user query (image or video and input text question) to the latent space the LLM model processes and generates a text output. While the research direction to communicate with the artificial intelligence (AI) models via natural language is promising, unfortunately, LLMs hallucinate outputs that might not be acceptable in many computer vision (CV) scenarios [9]. Within this thesis, we focus on the CV FM architectures and discriminative tasks: image classification and instance segmentation.

1.1 Motivation

It is estimated that by 2040, computing devices will require more energy than the world's production can provide [10]. The new FM are scaled to tremendous sizes and require enormous computing power for training and inference. For example, the BLOOM-176B model was trained on 384 NVIDIA A100 graphic processing units (GPUs), estimated to be equal to 24.7 tons of CO₂ [11]. Moreover, LLAMA 3.1 405B model was trained on 16000 NVIDIA H100 GPUs, which emitted 8,930 tons of CO₂ [7]. While training is the most compute-intensive and challenging

stage, the trained models are later utilized to respond to the user queries at inference. During this stage, the models require multiple dedicated accelerators to store their parameters and the user’s data. To tackle this problem, we can apply Approximate Computing (AxC) methods to compress the FM in order to reduce energy consumption, required memory, and latency while maintaining a similar level of predictive performance to the reference model.

One of the AxC methods is *quantization* [12]. This approach utilizes low bit-width data types in deep learning models for their parameters and intermediate activations. The smaller bit-width of the data type implies lower memory and processing requirements for DL models than the conventional data type used in the DL. Moreover, we can expect improved bandwidth transfer speed in accelerators like NVIDIA A100 or H100 GPUs with lower bit-width data types. Lastly, switching to integer instead of floating-point data types implies higher energy savings at the hardware level [13].

However, the application of quantization is not without its challenges. The trade-off between reduced precision and the predictive power of the DL model must be carefully considered. With lower bit-width, the models may be more susceptible to high quantization error. Moreover, while the quantization method might be effective for one trained model, it might not be transferable to another architecture. As a result, the search for the optimal bit-width for the model is a complex and challenging task. In this thesis, we take on the post-training quantization of the novel architecture - Vision Transformer without retraining.

1.2 Objectives

The main objective of this dissertation is to tackle the problem of post-training quantization of Vision Transformer models. This architecture is a novel type of DL model for CV that provides an attention mechanism that can capture local and global relationships between the objects presented in the input image.

The objectives of the thesis are:

1. Investigate the effect of the post-training quantization on the state-of-the-art models and explore methods to quantize them without high accuracy loss.
2. Propose novel algorithms for the quantization of Vision Transformers that quantize the model using static and dynamic quantization at the same time.
3. Develop open-source GPU kernels for quantized operations for the Vision Transformer architectures, allowing researchers and practitioners to develop new quantization methods more efficiently.

4. Design and implement mixed-precision attention that improves the throughput of the attention compared to the reference implementation.

1.3 Structure of the Thesis

The thesis is structured into six chapters, the overview of which is presented in Figure 1.1. In Chapter 2, we start by laying out the foundations of DL, CV, Transformers, and quantization, including a literature review and an inspection of the state of the DL frameworks and hardware.

In Chapter 3, we present the challenges in the post-training quantization of Vision Transformers. First, we thoroughly evaluate the effects of post-training quantization. Next, we analyze the signal-to-quantization-noise ratio of the intermediate activations and weights of these models. Lastly, we show that it is possible to apply mixed-precision quantization to achieve a consistent compression ratio with low quantization error.

Next, in Chapter 4, we propose a novel quantization algorithm - Hybrid Quantization. It allows the mixing of static and dynamic quantization within the Vision Transformer model. We present the algorithm for automatic selection based on the signal-to-quantization-noise ratio metric. Next, we evaluate the results of an extensive study of the effect of Hybrid Quantization on top-1 accuracy and inference latency on image classification tasks.

Lastly, in Chapter 5, we present an open-source framework for mixed-precision quantization of Vision Transformer models. First, we describe the quantized matrix multiplication and propose mixed-precision attention. Subsequently, we introduce the QAttn framework and its integration with PyTorch quantization workflows. Finally, we provide experimental results for the quantized matrix multiplication, mixed-precision attention, image classification with quantized Vision Transformers, and instance segmentation with the Segment Anything Model.

1.4 List of publications

In this dissertation, we focus on the post-training quantization of Vision Transformers, which resulted in four publications presented at conferences. We extended and adapted the text, figures, and tables from following list of publications:

1. Kluska P., Scheidgger F., Malossi A.C.I., & Quintana-Ortí, E.S., (2024). Beyond Static and Dynamic Quantization - Hybrid Quantization of Vision Transformers. British Machine Vision Conference (BMVC) 2024. Glasgow, United Kingdom.

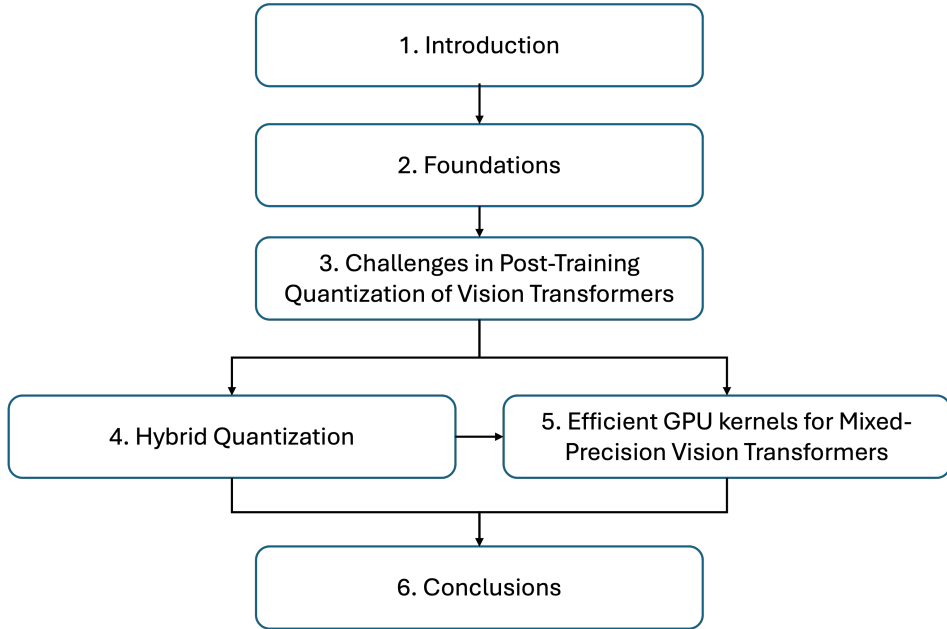


Figure 1.1: Overview of thesis structure

2. Kluska, P., Castelló, A., Scheidegger, F., Malossi, A.C.I., & Quintana-Ortí, E.S., (2024). QAttn: Efficient GPU Kernels for Mixed-precision Vision Transformers. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 3648-3657). Seattle, Washington, USA.
3. Kluska, P., Castelló, A., Scheidegger, F., Malossi, A.C.I., & Quintana-Ortí, E.S., (2023). Efficient deep learning inference models for low-energy applications Quantization of Vision Transformers, Mindtrek 2023, Doctoral Consortium, Tampere, Finland.
4. Kluska P., Scheidegger F., Malossi A.C.I., & Quintana-Ortí, E.S., (2023). Challenges in post-training quantization of Vision Transformers. In LLM4AI ACM SIGKDD International Conference on Knowledge Discovery and Data Mining 2023. Long Beach, California, USA.

Chapter 2

Foundations

In this chapter, in Section 2.1 and Section 2.2 we introduce deep learning (DL) and computer vision (CV), respectively, differentiating between downstream tasks such as image classification, object detection, and segmentation. We also provide an overview of the deep neural network architectures used in computer vision - convolutional neural networks (CNNs) in Section 2.2.4 and vision transformers (ViTs) in Section 2.3. Next in Section 2.4, we explain the concepts of quantization and integer inference, followed by a Section 2.5 review of the state-of-the-art methods used to compress the deep neural networks. Finally, in Section 2.6 we examine the deep learning frameworks and hardware used to accelerate training and inference.

2.1 Deep Learning

Deep learning (DL) is a subset of machine learning (ML) that allows programs to learn from data without being explicitly programmed [14, 15, 16]. DL utilizes neural networks that are built by stacking multiple layers together. Deep neural networks are more complex, larger, and more compute-intensive than other machine learning algorithms. On the contrary, DL models are trained end to end, that is the models learn the features from the processed data. While machine learning can handle many tasks, DL is particularly powerful for domains such as CV and natural language processing (NLP).

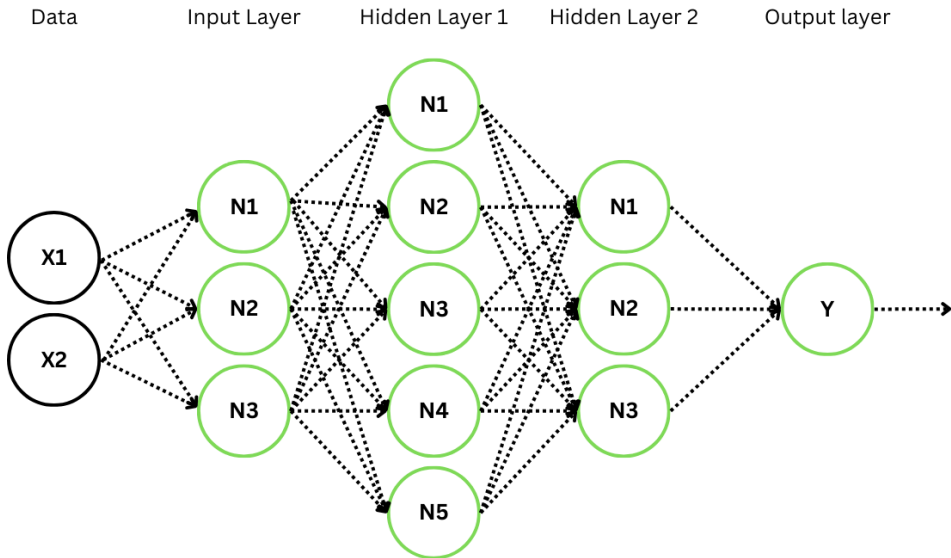


Figure 2.1: Example deep neural network architecture. This network has two inputs with three input neurons. Followed by two hidden layers with five and three neurons, respectively. Lastly, the output layer consists of single neuron.

Feedforward deep neural networks can be represented as a directed acyclic graph, as shown in Figure 2.1. The signal passes through the network from the input layer (left) to the hidden layers and finally to the output layer (right). The task of the deep neural network (DNN) is to map the input X to some output Y by applying the function $Y = f(X; W)$. For a classification task, the neural network maps the input X to category Y with learned parameters W . As mentioned previously, the network is constructed by stacking the layers, which can be treated as individual functions $f^1, f^2, f^3, \dots, f^n$, where f^1 is called the input layer, f^2 - second layer or hidden layer, and so on. The last layer, f^n , is called the output layer. The number of layers is a parameter of the network called depth.

Within the neural network, we distinguish many parameterized layers (e.g., linear, convolution), normalization layers (e.g., batch normalization [17], layer normalization [18]), and activations (sigmoid, hyperbolic tangent, Gaussian error linear unit [19]). The parameterized layers dictate the memory and computational requirements of the model. Modern deep neural networks range from millions to billions of parameters. The number of parameters in the individual parameterized layer is called width. The model's parameters, also called weights, are learned

using a backpropagation algorithm, such as stochastic gradient descent [20]. The majority of the models are trained using supervised or self-supervised methods [21]. In a supervised setting, we have labeled data, and the DL model is a trained representation of the data for the downstream task. In the self-supervised method, we utilize the unlabeled data in defined tasks (e.g., next-word prediction or image reconstruction). By applying self-supervised learning, we expect the model to learn an underlying representation of unlabeled data that should help train the model in a supervised setting on labeled data.

Recent advances in software and hardware have reduced the training and inference time of DL models. Modern hardware is equipped with specialized computational units (also called tensor cores or neural processing cores) dedicated to matrix multiplication, which is the most dominant operation in DL. The linear layer in deep neural networks performs a matrix multiplication $\mathcal{C}$ between inputs X and learned weights W adding a bias b

$$C = \mathcal{C}(X, W^T) = X \cdot W^T + b, \quad (2.1)$$

where X is the input matrix, of size $m \times n$, and W is the weight matrix, of size $k \times n$, and b is the bias vector of size k . The individual entries of the $m \times k$ result matrix C in (2.1) are thus given by a linear combination between the entries in the rows of X and W :

$$c_{ij} = \sum_{p=1}^n x_{ip}w_{jp} + b_j. \quad (2.2)$$

2.2 Computer Vision

Computer vision is a subfield of ML that focuses on enabling machines to extract, process, and respond to visual data. In CV, the most basic form of data is an image. Depending on the sensor and processing, images may have a different number of channels. For example, grayscale images have one channel, while color images have three. Additional channels might carry more data, such as depth. We can represent an image as a three-dimensional tensor whose dimensions are influenced by the image's channel, height, and width. Each value contains the intensity of the corresponding channel. For color images, the channels may represent the intensity of red, green, and blue colors, abbreviated as RGB. In Figure 2.2, for example we present the color image's example and each color channel's intensity. Usually, the value of the intensity is stored as an unsigned 8-bit integer (UINT8), the intensity of the channel uses the full UINT8 range, that is from a minimum value of 0 to a maximum value of 255.

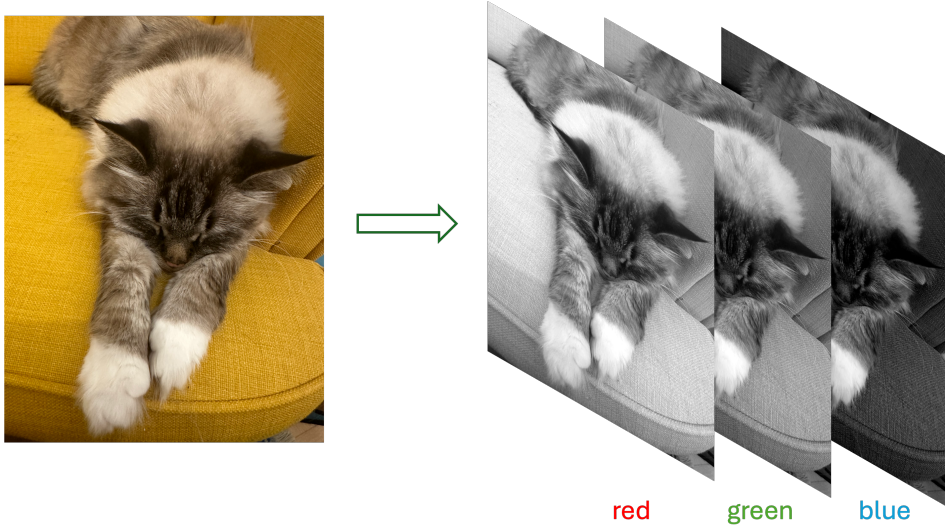


Figure 2.2: An example of an RGB (red, green, and blue) image with a cat.

In CV, we can distinguish tasks such as image classification, object detection, semantic segmentation, instance segmentation, and panoptic segmentation, among others. The overview of CV tasks is presented in Table 2.1.

2.2.1 Image Classification

Image classification is a task in CV where, given an input image, we assign a label to the most dominant object in the image. The DL models are trained using supervised methods with datasets containing the mapping of images to labels. The standard benchmark for image classification is ImageNet1K [22]. This dataset contains millions of images assigned to one of the thousands of classes that can be defined as “cat”, “chair”, or even dog breeds such as “husky” or “samoyed”. The DL model processes the input image and outputs a vector of real numbers called logits. The logits are then converted to the probability distribution using a Softmax function [15].

The metric used to evaluate the predictive power of the model is accuracy, also called top-1 accuracy. It is calculated as the ratio of the most likely predicted label to the ground truth label. It can be extended to top-k accuracy, where k is a positive integer, usually equal to 5, which extends the possibility that the model could predict the correct label in a broader range of predictions, five in this

Table 2.1: This table summarizes the computer vision tasks: image classification, object detection, semantic segmentation, instance segmentation, and panoptic segmentation. The table also shows the metrics typically used to evaluate the performance of these tasks.

Task	Description	Metric	
Image classification	Assign a label based on recognizing the content of an image.	Accuracy	
Object detection	Identify and locate objects in an image or video, and classify them into one or more object categories. A bounding box around the detected object can visualize the location in the image or video.	mean	Average Precision
Semantic segmentation	Identify and locate objects in an image or video. The predicted location should overlap with the object, allowing us to draw an overlay on top of the object. The detected objects are not distinguished.	mean	Intersection over Union
Instance segmentation	Similarly to semantic segmentation, identify and locate each instance of an object in an image or video, and classify each instance into a separate object category.	mean	Intersection over Union
Panoptic segmentation	Unifies semantic segmentation and instance segmentation.	Panoptic	Quality

example. However, the image classification limits the task to the assignment of the label to the image and just the most likely.

2.2.2 Object Detection

Usually, in real life, we find more than one object within the image and video feed. Moreover, image classification just predicts the object's label in the image while not predicting the object's location in it. In object detection, we predict the location and class of the objects in the picture. The DL model is trained to predict the object's coordinates, width, height, and label. The DL model's predictive power is expressed with mean Average Precision (mAP). This metric is calculated as a mean average precision over all dataset categories. To accept the prediction as a true positive in object detection, we select the threshold at the intersection over

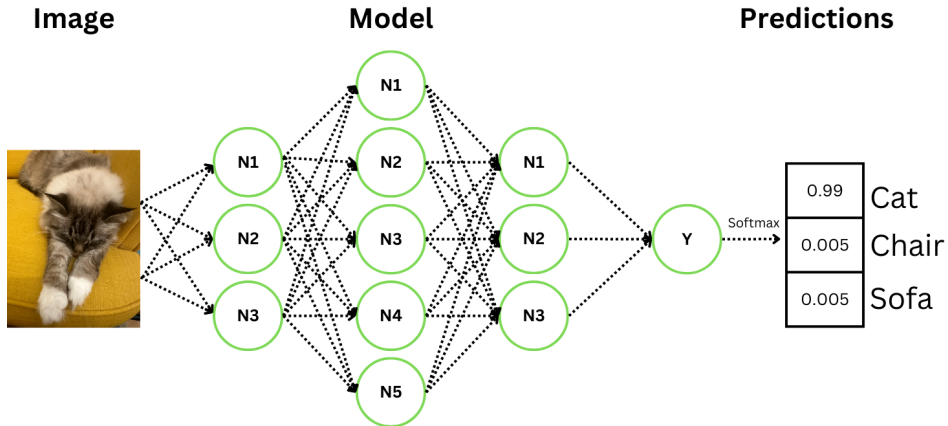


Figure 2.3: An overview of the image classification task. Given the input image, the DL model outputs the probability of an object in the image.

union (IoU), which is expressed as

$$\text{IoU} = \frac{B \cap B'}{B \cup B'}, \quad (2.3)$$

where B is ground truth bounding box and B' is the predicted bounding box or segmentation mask. The standard threshold for the IoU is 0.5. However, in datasets such as Common Objects in Context (COCO) [23], the mAP is calculated over all categories at various IoU thresholds - starting from 0.5 with a step size of 0.05 up to 0.95. The higher value of IoU means that the predicted bounding box overlaps better with the ground truth bounding box. Object detection is useful in scenarios where we would like to localize objects or people in the scene, e.g., in autonomous vehicles where we need to localize road signs, other cars, and pedestrians.

2.2.3 Segmentation

Segmentation is a fundamental task in CV. It involves dividing an image into its constituent regions or objects of interest, which allows for the isolation and analysis of specific features or patterns. This process helps to identify and separate different elements within an image, such as objects, textures, or regions. Unlike object detection, which aims to locate and classify objects in an image without considering their spatial extent, segmentation further identifies the precise boundaries and

shapes of these objects, providing a more detailed understanding of the visual scene. In autonomous driving, segmentation enables the identification of lanes, pedestrians, and obstacles to inform navigation and control systems. In comparison, object detection may only detect the presence of a pedestrian without identifying its exact location. Accurate image segmentation enables CV systems to extract meaningful information, reduce noise and ambiguity, and make more informed decisions.

We can distinguish three types of segmentation tasks: instance, semantic, and panoptic [24]. Instance segmentation focuses on identifying and separating individual objects or instances within an image, even if they belong to the same class or category. For instance, in a scene with multiple cars, instance segmentation aims to identify and segment each car individually rather than simply identifying the presence of cars. This approach is particularly useful in applications where individual object identification is crucial, such as in autonomous driving or robotics, where the system needs to track and interact with specific objects.

Semantic segmentation takes a more categorical approach, focusing on identifying and segmenting regions of an image based on their semantic meaning or class label. Semantic segmentation identifies all pixels belonging to the “car” class in the same scene with multiple cars without distinguishing between individual instances. This approach is beneficial in applications where understanding the overall scene composition is more important, such as scene understanding, image retrieval, or image generation.

While both instance and semantic segmentation share the goal of dividing an image into meaningful regions, they differ in their level of granularity and the type of information they provide, making them suited for different applications and use cases. The outputs of the segmentation we call masks, which are pixel regions with assigned class labels to the region; see Figure 2.4. To interpret the correctness of the masks, we can visualize them as an overlay on the image. However, we compute the mean Intersection over Union (mIoU) to compare the DL model’s performance as

$$\text{mIoU} = \frac{1}{N} \sum_{i=1}^N \frac{TP}{TP + FP + FN}. \quad (2.4)$$

Where for N classes it calculates a mean between ground truth annotated pixels (true positive (TP)) and the predicted pixels (TP , false positives FP , and false negatives FN).

Lastly, panoptic segmentation is a fusion of instance and semantic segmentation tasks. This technique aims to segment regions into semantic and instance groups jointly. The Panoptic Quality metric was introduced to evaluate the panoptic segmentation performance. Panoptic Quality is a composition of Segmentation

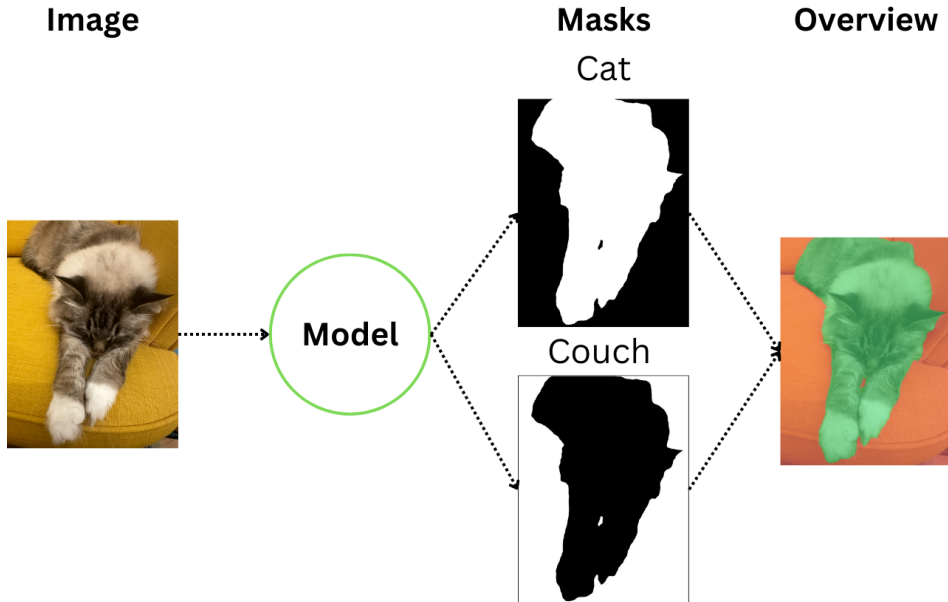


Figure 2.4: An overview of the segmentation task. The model predicts the masks and labels of the objects in the image. Then, the masks can be used to visualize the predictions as an overlay on the image. The white pixels in the mask show the pixel belonging to the class.

Quality and Recognition Quality. Segmentation Quality calculates the IoU of matched segmented objects, while Recognition Quality is computed as an F_1 score. The panoptic segmentation is helpful in scenarios where we need a comprehensive understanding of the scene and instance segmentation.

2.2.4 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of feedforward DNN [25]. They are created by stacking convolution layers interleaved with nonlinear functions (such as ReLU [26]) and normalization layers such as Batch Normalization (Batch-Norm) [17] or Layer Normalization (LayerNorm) [18]. The main operation of CNN is discrete convolution, which uses same-size filter banks trained with a backpropagation algorithm [14, 15]. Unlike fully connected layers, convolutional layers use shared weights, locality, and sparse connections, making them efficient

for image processing. The concept behind convolutional layers is that each filter learns a local pattern in the image that is activated regardless of its position in the image.

One of the first successful applications of CNNs was digit recognition. LeCun et al. created a five layer CNN that classified grayscale 28x28 pixel images with single digits [25]. However, the first multi-GPU training of the deep CNN was done by Krizhevsky et al [26], on the ImageNet1K dataset [22], they improved the top-1 accuracy by 8.2% compared to ML algorithms. AlexNet achieved 62.88% top-1 accuracy, while the modern CNN ConvNextV2 achieved up to 88.9% top-1 accuracy, depending on the model size [27].

CNNs have been successfully applied to image classification [26, 27], object detection [28, 29], image segmentation [30], and image generation [31, 15, 32]. However, this type of neural network has been shown to be more susceptible to texture bias than shape bias. This means that kernels trained on ImageNet1K are more likely to learn to recognize an object’s texture than its shape. In addition, the benefit of shared weights is also sensitive to a limited receptive field. The receptive field depends on the kernel stride and size. Nevertheless, we increase the receptive field of deeper convolution layers by stacking the convolution layers.

2.3 Vision Transformers

Vision Transformers [8] are a novel feedforward deep neural network based on the Transformer architecture from natural language processing [33, 11, 5, 6]. The base building blocks of Vision Transformers are multi-head self-attention (MHA) [34] and multi-layer perceptron (MLP). Initially, the input image $x \in \mathbb{R}^{H \times W \times C}$ is projected using a convolution layer and reshaped into patches $x_p \in \mathbb{R}^{N \times (P \times P \times C)}$. With H , W , and C we represent the input image dimensions - height, width, and channels, respectively. The patch size P is an effective size of the single patch of the ViT. Sequence length N is calculated as:

$$N = \frac{H \cdot W}{P^2}, \quad (2.5)$$

where the sequence length is proportional to the input image size but inversely proportional to the squared number of patches. The commonly used input image size and the number of patches in Vision Transformers are presented in Table 2.2 with the resulting sequence length (also called context length). Since the image is processed as a sequence of patches, the transformer architecture lacks an understanding of each patch’s spatial position within the image. To address this, positional embeddings are added to the image data, which encode the location of each patch.

2. FOUNDATIONS

These positional embeddings can be learned during model training or fixed as sinusoidal embeddings. Furthermore, similar to natural language processing, the learnable class token is prepended for the patches. The multi-head attention updates the class token to include the relevant embeddings to pass the calculated embeddings to the classification layer.

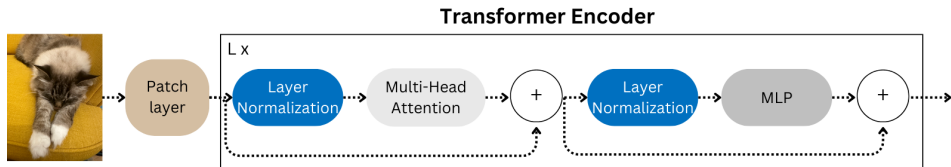


Figure 2.5: Vision Transformer Model Overview: This diagram shows the architecture of the ViT, which takes an input image, divides the image into patches, and processes them through a series of transformer layers to produce an output representation that can be used for various computer vision tasks. Each Transformer layer consists of Layer Normalization, MHA, and MLP blocks with residual connections in between. The illustration of the Vision Transformer architecture was inspired by Dostovitskiy et al. [8]

Processed patches are then fed to the transformer encoder, which consists of multiple stacked transformer blocks. Each block starts with layer normalization [18], and normalized latent activations are next processed by MHA. Subsequently, the residual connection [35] adds proceeding activations to the output that is next again normalized using layer normalization; see Figure 2.5. Lastly, the intermediate activations are processed by a MLP, a block consisting of two linear layers with GELU activation [19] in between.

Table 2.2: Sequence length N (eq. (2.5)) relative to image size $H \times W$ and patch size P . This table shows the relationship of patch size and image size to sequence length. Each sequence length is incremented by one because the class token is prepended to the sequence length.

		Image Size $H \times W$	
		224×224	384×384
Patch	8	785	2305
	16	197	577
Size P	32	50	145

Depending on the number of transformer blocks L , hidden size $\hat{D}$, MLP size f , and number of heads h , we can scale the size of the ViT; see Table 2.3. These

models come in various sizes, ranging from tiny (ViT-T), small (ViT-S), base (ViT-B), large (ViT-L), huge (ViT-H) up to gigantic (ViT-G). Furthermore, these models differ in the patch size - the larger patch size reduces the sequence length in the attention mechanism, resulting in a smaller computational footprint as presented in Table 2.2. Depending on the input image size I and patch size P , we can differentiate the models as ViT-X/P/I (Where X is the letter denoting the model size), e.g., ViT-L/16/224 or ViT-B/32/384.

Table 2.3: Architecture and parameter count of ViT models. This table provides an overview of the architecture and parameter counts for several ViT models, highlighting the variations in layer count, MLP size, hidden size, number of attention heads, and total parameters across models.

Model	Layers	MLP size f	Hidden size D	# Heads h	# Params
ViT-T [36]	12	768	192	3	5.7M
ViT-S [8]	12	1536	384	6	22.2M
ViT-B [8]	12	3072	768	12	86M
ViT-L [8]	24	4096	1024	16	307M
ViT-H [37]	32	3840	1280	16	630.8M
ViT-G [37]	48	8192	1664	16	1.8B

The MHA is a pivotal operation block in the transformer’s architecture. It consists of a projection linear layer that transforms latent variables for multiple heads h of head size d for the given sequence length N . The attention mechanism operates on three input tensors, namely query Q , key K , and value V , all of the same size, i.e., $Q, K, V \in \mathbb{R}^{h \times N \times d}$, where d is the head dimension. In Figure 2.6 we present the computational flow of MHA block in the Vision Transformer.

The attention module operates by multiplying the values matrix V with a weighted matrix, that is calculated as a function of the similarity between Q and K . Concretely, the similarity is determined by multiplying the matrices Q and K , and then scaling the result as follows:

$$\bar{E}_1 = \frac{\mathcal{C}(Q, K^T)}{\sqrt{d}}. \quad (2.6)$$

The attention weights are then obtained by applying the Softmax function

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^N e^{x_j}} \quad (2.7)$$

to the result:

$$E_1 = \text{Softmax}(\bar{E}_1), \quad (2.8)$$

2. FOUNDATIONS

and the resulting weights are used to adjust the values for the attention output:

$$E_2 = \mathcal{C}(E_1, V). \quad (2.9)$$

The computational complexity of the attention is $O(N^2 \times d)$ with space complexity of $O(N \times d + N^2)$, as we need to store and materialize the intermediate attention outputs. Due to that, many alternative implementations were proposed. FlashAttention [38] and FlashAttention2 [39] proposed an effective use of NVIDIA Ampere GPU memory hierarchy and input-output (IO) awareness to transfer the tensors from HBM memory to SRAM and utilization of L2 cache. Moreover, they provide a fused attention kernel that utilizes tiling, which loads blocks of matrices into the memory and computes the numerically accurate attention outputs. The designed kernels minimize L2 cache misses by avoiding the materialization of the intermediate attention matrix. As mentioned, the kernel only has access to a block (or tile) of the attention map. As a result, the online softmax is applied, which calculates the statistics for the softmax over these blocks and normalizes the block output, leading to accurate results. FlashAttention partitions the workload based on batch size and head dimension, while FlashAttention2 additionally parallelizes computations along the sequence length. Lastly, they avoid many elementwise floating point operations that could be computed only once. As a result, the FlashAttention additional space complexity equals $O(N)$ with the same space complexity of $O(N^2 \times d)$.

While FlashAttention utilizes IO internals of the GPU accelerator, Liu et al. propose Window and Shifted-Window attention introduced in the Swin Transformer model [40]. The Window attention does not focus on the global attention of the image as it is computationally expensive. It proposes dividing the patches into windows on which the attention is computed, thus reducing the complexity of attention mechanisms. In the succeeding transformer block, they propose shifted window attention, which moves the windows to enable attention to focus cross-window.

ViTs have demonstrated the ability to scale with model size and improved performance with pre-training data in self-supervised training. Yet, the pre-training step requires a large amount of data and computational resources. During this phase, the model is pre-trained on a much larger dataset - both in terms of the number of images and the variety of classes - compared to the target dataset. Once pre-training is complete, the original classification head of the ViT is removed and replaced with a linear layer initialized to zero. The model is then fine-tuned on the downstream dataset. As a result, the Data efficient Image Transformer (DeiT) has been proposed [36]. DeiT shared the architecture of the original Vision Transformer and was trained without pre-training. However, it included more data

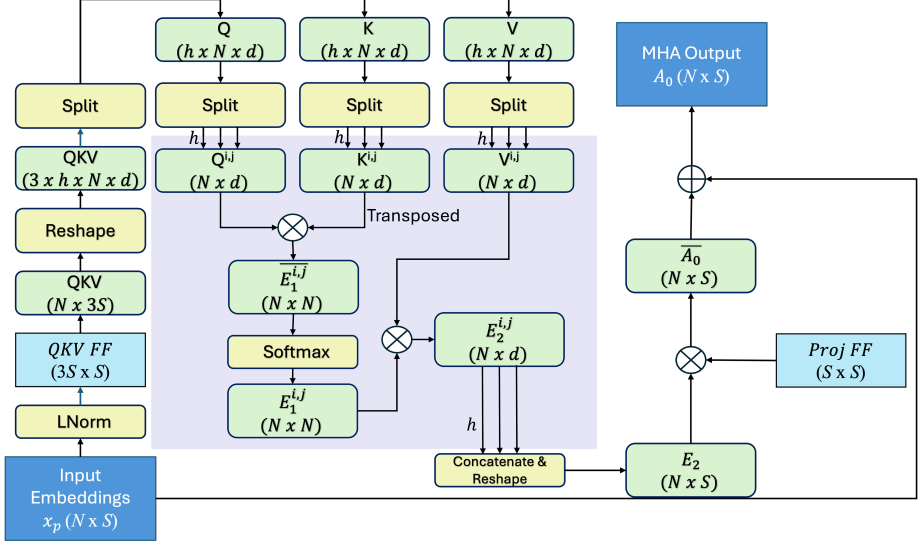


Figure 2.6: This figure illustrates the MHA block within ViT encoder.

regularization and introduced the distilled knowledge trained models from the CNN teacher. Building on DeiT, Houvrou et al. proposed DeiT3 [41], which rethought the training procedure of the ViTs. In DeiT3, they introduced LayerScale [42], reduced augmentation during training, and low-resolution image pre-training to reduce computational requirements.

2.4 Quantization

The standard data type in deep neural networks is IEEE 32-bit floating point (FP32) [43]. This type reserves 8 bits for the exponent, 23 bits for the significand, and 1 bit for the sign, for a total of 4 bytes. The FP32 data type has the capability of representing huge dynamic ranges of values. Henceforth, algorithms that rely on large dynamic ranges tend to work better with this type. Note that strictly FP32 is known to have many flaws, but most of them are not relevant in practical applications. As shown in Figure 2.7, increasing the size of the DL model (number of parameters) increases the DL model's performance on the downstream task, in this case, image classification. However, to store the parameters of the ViT-H model [37], we need more than 2.5 gigabytes (GB) of memory without considering intermediate activations and computational requirements. To alleviate this problem, we can use quantization [44, 12].

2. FOUNDATIONS

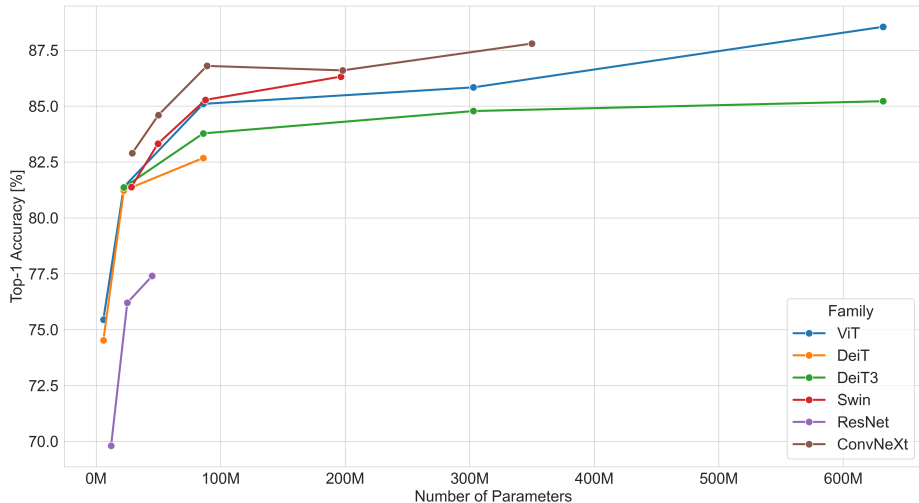


Figure 2.7: Predictive power (Top-1 Accuracy) vs. Number of Parameters for Image Classification DL Models on ImageNet1K [22] This figure shows the relationship between number of parameters and performance (Top-1 Accuracy) on ImageNet1K for several state-of-the-art image classification models, including Vision Transformers (ViT) [8], Data Efficient Image Transformers (DeiT [36] and DeiT3 [41]), Swin Transformers [40], ResNets [35], and ConvNeXt [27], demonstrating the trade-offs between model size and performance.

Quantization is a technique that reduces the required bit width of the weights and/or activations of the DL models. Modern DL accelerators support floating point formats such as IEEE 16-bit floating point (FP16) [43], Brain Float 16-bit (BFloat16) [45], and NVIDIA’s TensorFloat32 [46]. Moreover, accelerators also support an 8-bit floating point (FP8) with two representations depending on the precision required [47]. Table 2.4 shows the differences between the various floating-point data types and the differences in the bit widths used for the exponent and mantissa. By switching from FP32 to FP16 or BFloat16, we have half the memory requirements for the DL models, and we can use specialized computational units - called tensor cores - to speed up matrix multiplication. The reduced memory also allows us to process larger amounts of data and transfer it to the accelerator.

While reducing the number of bits can diminish memory consumption, the floating-point arithmetic units in accelerators are, unfortunately, more complex in logic structure and less power-efficient than integer arithmetic units [48]. According

Table 2.4: Floating point datatype formats. This table lists the bit configurations for various floating-point data types, including the number of bits allocated to the sign bit, exponent, and mantissa, and illustrates the tradeoffs between precision and storage requirements.

Data type	Sign Bit	Exponent	Mantissa
Floating-Point 32-bit (FP32) [43]	1	8	23
Floating-Point 16-bit (FP16) [43]	1	5	10
BFloat16 [45]	1	8	5
TensorFloat32 [46]	1	8	10
Floating-Point 8-bit (FP8) [47]	1	4	3
Floating-Point 8-bit (FP8) [47]	1	5	2

to van Baalen et al., the FP8 data type is at least 50% less efficient than INT8 [48].

To convert an FP32 to INT8 or UINT8 value, we can apply the linear quantization function $\mathcal{Q}$ as

$$X_q = \mathcal{Q}(X) = \lfloor \frac{X}{s} \rfloor - z, \quad (2.10)$$

where s is the scaling factor computed for the given tensor, e.g., as an absolute min-max value in the range of the tensor divided by the integer data type range q_{max} and q_{min} (see Table 2.5)

$$s = \frac{\max(\text{abs}(X))}{(q_{\max} - q_{\min})/2}. \quad (2.11)$$

Where abs is an element-wise operation on the matrix, while the $\max$ returns the scalar maximum value from it. The inverse function to the quantization is dequantization, which retrieves an approximate floating point value from the quantized value

$$\hat{X} \approx \hat{\mathcal{Q}}(X_q) = s(X_q + z). \quad (2.12)$$

As quantization is an approximation method, the resulting floating-point value will be inexact to the reference value. The difference between those two values is called quantization error, which can be computed using the Euclidean norm. Moreover, the quality of the quantization signal through the network can be measured using the signal-to-quantization-noise ratio (SQNR) metric

$$\text{SQNR}(X, Y) = 20 \log_{10} \frac{P_S}{P_N} = 20 \log_{10} \frac{\|X\|}{\|X - Y\|} [dB], \quad (2.13)$$

2. FOUNDATIONS

where P_S and P_N respectively represent the signal’s power and the quantization noise’s power. This metric measures the signal quality between the quantized and the floating-point reference operation. A higher SQNR value represents better signal quality, meaning the quantization error is less pronounced, and the quantized operation is closer to its full-precision baseline. We use the Euclidean norm to calculate the maximum amplitude and the difference between the original and quantized signals.

The quantization scheme in Equation (2.10) is categorized as affine or asymmetric. It is helpful when the values in the tensor are skewed, and we need to shift the zero point z to truly represent the floating-point zero value. However, if we set the $z = 0$, then we are left with symmetric quantization, where negative and positive values have equal quantized ranges

$$X_q = \mathcal{Q}(X) = \lfloor \frac{X}{s} \rfloor. \quad (2.14)$$

Table 2.5: Integer data type formats. This table lists the formats and value ranges of several integer data types, including the number of bits, minimum value, and maximum value for each type, highlighting the tradeoffs between precision and storage requirements.

Data type	# Bits	Minimum Value q_{min}	Max Value q_{max}
INT32	32	-2^{31}	$2^{31} - 1$
INT16	16	-2^{15}	$2^{15} - 1$
INT8	8	-128	127
UINT8	8	0	255
INT4	4	-8	7

In DL, quantization methods can be divided into two main subgroups - quantization-aware training (QAT) and post-training quantization (PTQ) [12]. QAT requires a dataset on which the quantized model is trained. On the one hand, it usually results in a higher predictive power of the quantized model for the downstream task. On the other hand, access to the training dataset and computational resources is required to train the model from scratch. The backpropagation algorithm must be modified to include the straight-through estimation (STE) step to compute gradient updates [49]. In addition, the training procedure introduces overhead by applying quantization and dequantization operations to the weights and activations. Nevertheless, the QAT method might suffer from weight oscillation during training, i.e., the trained parameter oscillates between two quantization levels, which may impact the performance of the quantized model [50]. Moreover, the

STE method is an approximate identity function that might inject noise into the computed gradients. Alternatively, post-training quantization is a domain where the task is to quantize an already trained model from a floating-point range to an integer counterpart. Within the post-training quantization, we differentiate static quantization and dynamic quantization.

2.4.1 Static Quantization

Static quantization of the DL model requires a calibration dataset. Depending on the downstream task, the calibration dataset consists of hundreds to thousands of samples. Those samples are used to estimate the scaling factor s and zero point z of activations at each layer f of the model. The activation observers collect statistics on the inputs and outputs of the given node. After the calibration step, the observers are removed from the model's graph, and the scaling factors and zero points are set for the given node for the model's lifetime. The quantized DL model nodes then accept a quantized tensor and produce a quantized tensor. Figure 2.8 shows the process of calibration and quantization of a node of the DL model.

Static quantization benefits from reduced bit-width as we can move more data at higher speeds with the same memory bus. Moreover, integer operations consume less energy than their floating-point counterparts [13]. However, the calibration set may not capture the dynamic range of the data, resulting in the information in the input data being clipped.

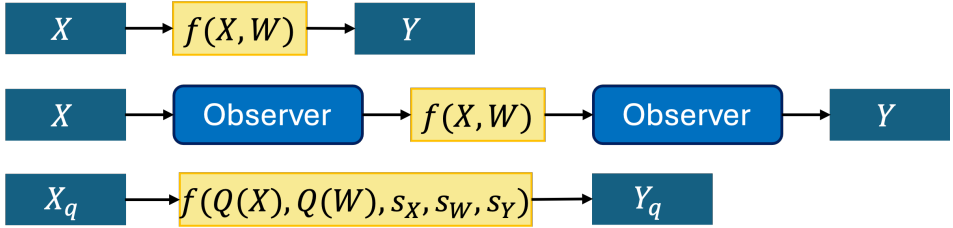


Figure 2.8: Static post-training quantization of DL model nodes. This figure shows an example of the node being quantized using static quantization. First, we insert the observers that compute the statistics of the input and output of the node. After calibration the observer nodes are removed and scaling factors and zero points (removed for brevity) are inserted into the function as a constant parameters.

To perform matrix multiplication $\mathcal{C}$ Equation (2.1) involving integer arithmetics, we require the quantized tensors $\mathcal{Q}(X)$ and $\mathcal{Q}(W)$, scaling factors for input s_X ,

weights S_W , and outputs S_C .

$$C^S = \mathcal{Q}(\mathcal{C}(\mathcal{Q}(X), \mathcal{Q}(W^T))) \quad (2.15)$$

so that, for the entries of the result,

$$c_{ij}^S = s^S \sum_{p=1}^n \mathcal{Q}(x_{ip}) \mathcal{Q}(w_{jp}), \quad (2.16)$$

where s^S is a static re-quantization scaling factor that is computed as a linear combination of the scaling factors of X , W and C [51]:

$$s^S = \frac{s_X s_W}{s_C}. \quad (2.17)$$

The intermediate output of the static quantized matrix multiplication is stored with higher precision than the input tensors - usually INT16 or INT32 - to avoid overflows. The scaling factor s^S is used to re-quantize the output to the desired bit-width, e.g., INT8.

2.4.2 Dynamic Quantization

Unlike static quantization, dynamic quantization does not require a calibration data set or step. The parameters of the DL model are quantized to lower precision, while the unparameterized operations - activation layers, normalization layers, skip connections - are kept in floating-point arithmetic. In Vision Transformers, only the linear layer is dynamically quantized, transforming it into a matrix multiplication operation

$$C^D = \mathcal{C}(\mathcal{Q}(X), \mathcal{Q}(W^T)), \quad (2.18)$$

with its output entries given by

$$c_{ij}^D = s^D \sum_{p=1}^n \mathcal{Q}(x_{ip}) \mathcal{Q}(w_{jp}). \quad (2.19)$$

The quantization parameters - scaling factor s_X is calculated while processing the input data to the given layer, while the scaling factor s_W is precomputed

$$s^D = s_X s_W. \quad (2.20)$$

Moreover, due to the lack of output quantization parameters, the output of the matrix multiplication is a floating-point tensor. Dynamic quantization achieves a

compression similar to static quantization. Nevertheless, the performance is slightly decreased due to the on-demand quantization of the input to the linear layer. One of the alternatives to dynamic quantization is weight-only quantization, which similarly quantizes to lower bit-width parameters of the DL model. However, the weights are dequantized to floating-point data type before the matrix multiplication operation is performed.

2.5 Related Work

In recent years, there has been a noticeable increase in the development of large foundational models [52], pre-trained on massive amounts of data [37, 5, 11, 53]. Within natural language processing and computer vision, these models have demonstrated the ability to scale up to 176 billion (176B) [54, 11] parameters for natural language processing (NLP) and 22 billion parameters for computer vision [55]. The foundational models are then fine-tuned for tasks such as text or image generation [56, 57, 58], document retrieval [59], image classification [37, 55], and instance segmentation [60]. However, using these deep learning models in inference requires significant computing and memory power. In this section, we review several compression methods for DL models, starting briefly with knowledge distillation [61] and mixture of experts [62]. Followed by sparsification [63] and quantization [12].

2.5.1 Knowledge distillation

The concept of knowledge distillation [64] is built upon the student-teacher framework. This approach aims to achieve comparable accuracy from a smaller deep learning model to a larger model already trained on the specific task. However, it is essential to note that this process involves training the models from scratch. In response-based knowledge distillation, the student network is optimized using both the downstream task loss and the response of the teacher network [64]. This guides the student network to replicate the teacher’s responses. In feature-based knowledge distillation, the distillation loss is calculated between respective layers or blocks of layers between the teacher and the student [65, 66]. The most straightforward approach to knowledge distillation is offline training, i.e., the teacher model weights are frozen. However, there is also ongoing research in online knowledge distillation that also updates the parameters of the teacher model. Moreover, with closed-source datasets and open-source models, there is an opportunity to distill the knowledge of models to improve the performance of the smaller model. AM-RADIO [67] distills the knowledge from CLIP [68], DINOv2 [69], and SAM [60].

As a result, it improves the feature representation and performance of downstream tasks.

2.5.2 Mixture of experts

The ensemble of machine learning models includes multiple models trained for a specific task to improve their predictive ability compared to a single model [70]. This concept has been adapted to deep learning as a mixture of experts, especially in natural language processing. The mixture of experts consists of multiple models, usually with the same architecture but with fewer parameters than a single large foundation model, for example, a mixture of 7B models [62]. A simple gating function, or “router”, selects the experts based on the input for processing. This simplifies the training process, as each expert can occupy its accelerator without distributing the model between accelerators. While the mixture of experts reduces inference time by running multiple smaller models, it does require all the models to be loaded into memory. For example, Mixtral 8x7B [71] requires storage equivalent to a 47B parameters model, but it only utilizes 12B parameters for inference.

2.5.3 Sparsification

Large deep learning models are often over-parametrized, with some neurons and layers learning model noise instead of valuable data. Sparsification [63] aims to address this issue by removing unnecessary weights from the model. There are two types of sparsification: unstructured and structured. Unstructured sparsification allows for randomly removing weights, filters, channels, or entire layers without restriction. However, when using unstructured pruning, it is necessary to encode the positions of the retained weights. This can be achieved through run-length encoding, compressed sparse row (CSR), compressed sparse column (CSC), and coordinate offset, depending on the specific format and sparsification technique. In high-performance computing, CSR and CSC are commonly employed. These methods represent the matrix using three vectors: the value vector, the row vector, and the column vector. The row and column vectors store the indices of the values. In CSR, the rows are accessed first, while in CSC, the columns are accessed first.

Unstructured pruning might decrease the model’s required storage, but using it in modern accelerators is challenging [72]. With structured pruning, we aim to remove the blocks of weights, layers, or filters in adjacent locations. In recent NVIDIA GPU accelerators, the tensor cores allow for 2:4 structured pruned matrix multiplication operations starting from the Ampere generation [46]. It means that weights are grouped into four blocks with two values set to zero. However,

while pruning is an interesting research direction, Kuzmin et al. have shown that quantization is generally better than unstructured and structured pruning [72].

2.5.4 Quantization

Convolutional Neural Networks

In convolutional neural networks, various methods exist for quantizing models to 8, 4, 2, or even 1 bit [12]. Post-training quantization methods generally achieve up to 4 bits, whereas two-bit or binary networks are typically obtained through quantization-aware training. Within post-training quantization, the common method to accelerate inference of the convolutional neural networks is batch normalization folding, i.e., batch normalization statistics are fused into preceding convolutional layer weights [51, 44]. Moreover, usually in those networks, RELU is used as nonlinearity, which can efficiently be fused into a convolution kernel to avoid re-quantization operation before it. Generally, most convolutional networks are statically quantized to INT8 except for those containing depth-separable convolutions, e.g., MobileNetV2 [73]. It was shown that the channels have varying ranges within depth-separable convolutional layers. As a result, quantization clips small values. In order to refine that, cross-layer equalization was proposed, which rescales the channel from one layer and applies the scaling factor to the succeeding layer to equalize the weights in the layer [74].

Moreover, it was shown that adding bias to the output of the cross-layer equalized convolutional layer might result in higher dynamic ranges. Similar to cross-layer equalization, the next layer might absorb the bias. Nagel et al. show that when converted to INT8 the MobileNetV2 model, achieving 71.32% top-1 accuracy on ImageNet1k, loses its predictive power completely. After applying cross-layer equalization, the performance recovers to the quantized model within 2% of the reference. After absorbing the bias, the top-1 accuracy achieved 70.92% on the image classification task [74].

The most challenging task in the quantization of convolutional neural networks is clipping the high values in activations or rounding small values to the same integer value. Banner et al. propose the ACIQ method [75], which analytically calculates the quantization ranges for activations and weights. Furthermore, they study bit allocation per channel to optimize the storage requirements. However, bit allocation is problematic when implemented on the deployment hardware. Similarly, Nagel et al. observe that the nearest integer rounding might be suboptimal in quantization and propose an adaptive rounding method (AdaRound) [76]. AdaRound, based on the samples, optimizes the weights to round the value to decide the appropriate integer value that minimizes the loss function. With this method, the authors

could quantize ResNet50 [35] to 4 bits for weights and 8 bits for activations while keeping the model’s performance within 1% of the floating-point reference.

In order to use a lower quantization bit-width, it is necessary to undergo quantization-aware training. Similarly to post-training quantization, we would start from the pre-trained model. However, in this case, it is possible to start with random weights. The methods build on the work of Bengio et al [49]. by applying a straight-through estimation backpropagation algorithm. With extreme quantization (1 or 2 bits per layer), the authors of BinaryNet, XNOR-Net [77], and XNOR-Net++ [78] showed that it is possible to achieve a heavily compressed model based on AlexNet [26] that achieves satisfactory accuracy in image classification.

Moreover, with XNOR-Net, the matrix multiplication can be replaced with XNOR and bitcount operations, which are around 58 times faster on the CPU. Considering Internet of Things (IoT) devices, Lin et al. propose an MCUNet [79, 80] that can run on a device even with 256KB of SRAM while achieving better accuracy than XNOR-Net [81]. They achieve that by utilizing neural architecture search to find the optimal architecture for the IoT and developing TinyEngine to run those networks efficiently.

Esser et al. provide a Learned Step Size Quantization (LSQ) method that builds on a straight-through estimator [82]. They introduce a method that can learn quantization steps alongside the network parameters. This lets the network learn the optimal parameters for the given bit width. Bhalgat et al. extend the LSQ to LSQ+ [83] to include the newer activation functions like Swish [84] and Mish [85]. Both methods allow the network to be trained for various bit widths - 8, 4, 3, and 2.

Moreover, the utilization of one of the compression techniques does not exclude the usage of another one. Polino et al. [86] explored a knowledge distillation from the teacher to a quantized 2-bit or 4-bit student network. With Quantized Knowledge Distillation, they can achieve near teacher performance (ResNet50) with two student ResNet34 networks at INT4.

Transformers

Large language models (LLMs) are transformer architectures scaled up to extremely large sizes, such as 7B, 30B, 70B, 176B, and 340B, for natural language processing tasks [11, 5, 6, 54, 87, 88, 89]. These models require significant memory and computing power. With those model sizes, quantization-aware training is almost impossible due to the complexity of training those models. The Quantized Low-Rank Adapters were thus proposed as a proxy to quantization-aware training, where the LLM is compressed to 4 bits and frozen while adapters are updated [90]. Therefore, ongoing research is being conducted on weight-only quantization to

reduce the memory requirements of these models. This method helps deploy these models on memory-constrained devices like consumer-grade GPUs or dedicated accelerators.

Frantar et al. introduced a post-training quantization method called GPTQ [91]. This method is designed for LLMs and involves computing Hessians of the weights. They use a greedy search to determine the next layer to quantize, and within the layer parameters, they perform quantization per row. Their method demonstrates minimal perplexity increase compared to the floating point reference. Perplexity is a metric used to evaluate the ability of the NLP model to predict the test data. The lower the perplexity value, the less confused the model is about what the next predicted word should be. Additionally, they could run large models with up to 176B parameters on a single data center GPU with 80GB memory using 4-bit or 3-bit weights.

Dettmers et al. discovered that quantizing all weights and activations in LLMs is challenging [92]. They recommended isolating and storing the outliers as a 16-bit floating point (FP16). In contrast to the GPTQ method, their approach involves quantizing weights and activations to INT8 while preserving outliers in FP16. They divide the tensors into INT8 and FP16, perform INT8 matrix multiplication with an accumulator set to INT32, and then dequantize the outputs to FP16. These outputs are then combined with the outlier matrix multiplication results. Additionally, they emphasize vector-wise quantization, which offers better quantization granularity but requires additional scaling factors. Nonetheless, their method achieves results comparable to the reference floating point OPT models.

Dettmers and Zettlemoyer examined the ideal model size and bit-width for quantizing a model in a zero-shot manner [93]. They conducted experiments with models ranging from 19 million to 176B parameters using 3-bit to 8-bit quantization. They explored integer and floating-point quantization and determined that the most optimal zero-shot quantization is a 4-bit floating-point approach with a block size equal to 128. One scaling factor is kept for every 128 values in the tensor matrix. However, they noted that no accelerators were available for 4-bit floating-point at the time of their study, so they recommended using an integer quantization to reduce inference latency.

Lin et al. proposed an Activation-Aware Weight Quantization (AWQ) technique for the quantization of LLMs [94]. Similar to Dettmers et al. [95], they observed that not all weights in the LLMs layers are equal and essential. They focused on inspecting the activations to determine the optimal per-channel quantization of the weights. Their work concentrated on weight-only quantization with a block size of 128 and INT4 data type. Since they were concerned only with memory issues, the computation was done in FP16. They demonstrated speedups on consumer-grade GPUs and the Jetson Orin of up to 3.9 and 3.5 times, respectively.

2. FOUNDATIONS

Vision Transformers are inspired by the architecture of the encoder module of LLMs. However, they handle different modalities - images. Moreover, modern ViTs are much smaller than LLMs, achieving high accuracy on the downstream task even without reaching 1B parameters. This opens an opportunity to quantize the model’s weights and activations to deploy them on edge or mobile devices and small accelerators.

Lin et al. observed that the methods that worked to quantize the convolutional neural networks, unfortunately, do not work anymore on the vision transformers [96]. Similar to researchers exploring the quantization of LLMs, they have found outliers in activation maps of layer normalization and attention operation. To enable integer-only inference, they propose a power-of-two factor for LayerNorm, enabling every channel of LayerNorm individual scaling factors. Moreover, they modified Softmax, which would have to run with floating precision, with an approximate version - Log-Int-Softmax. As a result, they could quantize the model’s weights, activation, and attention maps to INT8 within 1% accuracy degradation.

Following the LLMs quantization, Liu et al. employ mixed-precision quantization, i.e., they leave out Layer Normalization and activations in floating-point precision while applying quantization to the linear layers and matrix multiplications in attention layers [97]. They define the nuclear norm to optimize the attention module’s appropriate scaling quantization factor for matrix multiplication. Moreover, they define ranking loss to achieve consistency of activations order after quantization. However, the method is not one-shot post-training quantization as they are defining search optimization algorithms to find optimal scaling quantization factors for linear layers. Nevertheless, they can achieve negligible top-1 accuracy loss on the ImageNet1k dataset for ViT and DeiT models in their mixed-precision setup. PTQ4ViT is a method that explores the Hessian-guided optimization of linear layer’s weights and activations of matrix multiplication of attention modules [98]. Moreover, the authors find that the activation maps of Softmax and GELU activations are hard to quantize and propose a two-factor quantization. They divide activation map regions into two regions with separate quantization scale factors. As a result, the quantization values are closer to the reference floating-point activations. They showcase that with their method, they can utilize small amounts of images, i.e., 32 and 128 while achieving negligible accuracy degradation.

QuantFormer has developed a method for training and fine-tuning models using 2, 3, and 4 bits [99]. They focus on linear layers and attention, exploring layer-wise and channel-wise quantization differences. Additionally, they propose patch-aware group-wise quantization, where up to 16 groups are selected for the weights to maintain quantization parameters. In their training setup, they obtained top-1 accuracy degradations of 14.7%, 4.5%, and 1.7% for DeiT-S using 2, 3, and 4 bits,

respectively.

As most quantization research focuses on linear layers and attention modules, the authors of I-ViT focus on integer approximations of activations and Layer Normalization [100]. They propose Shiftmax instead of Softmax, ShiftGELU instead of GELU, and I-LayerNorm instead of LayerNorm. They argue that their method allows for integer-only inference without performing those activations with a fake quantization mechanism. They show minimal accuracy degradation compared to the floating-point reference ($< 1\%$) for ViT, DeiT, and Swin models. Moreover, they showcase up to 4 times faster inference on consumer-grade GPU compared to the baseline. Similarly Zhang et al. [101] proposed an integer GELU, LayerNorm, and Softmax activation using INT32. They have evaluated the ViT architecture on the Apple M1 and A13 chips achieving up to 5 times speedups in kernel execution.

Instance-aware group quantization [102] dynamically allocates activations of linear layers and softmax input activations to appropriate groups that would incur low quantization error. They utilize Kullback-Leiber divergence loss between the quantized and baseline models' predictions at each layer. However, except for the grouping algorithm for the Softmax, which boosts the quantized predictive power, authors do not provide a quantized implementation of the Softmax activation. Moreover, similar to others, they do not consider the quantization of the LayerNorm layers. Nevertheless, their experiments showcase that with up to 12 groups, the model's weights and activations can be compressed up to 4 bits with 19.46%, 7.78%, 5.22% top-1 accuracy degradation for ViT-T, ViT-S, and ViT-B, respectively.

GPUSQ-ViT explores the possibility of combining knowledge distillation, 2:4 sparsity, and quantization in one workflow [103]. The authors explore the sparsification of the weights and later quantization to INT8 and INT4 to accelerate the matrix multiplication on the NVIDIA GPUs. Their method can boost the inference speed by up to 3.43 times throughput an NVIDIA A100 GPU with model size reduced by up to 12.7 times.

2.6 Deep learning frameworks and hardware

Recent advances in deep learning have sparked the development of specialized computing units within accelerators called Tensor Cores in NVIDIA GPUs [46, 104, 105], Neural Engine in Apple Silicon [106], Google's Tensor Processing Units (TPU) [107], and many more [108]. The task of those specialized compute units is to accelerate matrix multiplication in deep learning applications, which is the fundamental operation both in convolutional neural networks and transformers. However, researchers and developers must use specialized application programming interfaces (API) to utilize those. This section describes some frameworks and

hardware used for deep learning.

One of the most popular frameworks used to accelerate deep learning models is CUDA [109]. CUDA is a proprietary platform from NVIDIA that accelerates parallel processing on GPUs. This runtime extends C language with the CUDA API and LLVM compiler [110]. This allows the writing of unified code for both the CPU and GPU. Moreover, NVIDIA has released a closed-source library - CUDA Deep Neural Network (cuDNN) [111] that contains kernels to accelerate both inference and training of the deep learning models. Dedicated accelerators for those frameworks are NVIDIA GPUs. The first generation of Tensor Cores was introduced in the Volta architecture [112], allowing mixed-precision training and inference in FP32 and FP16. Ampere’s Tensor Core additionally enabled the acceleration of matrix multiplication in INT8 and INT4 as well as BFloat16 and TensorFloat32 [46]. With foundation models requiring extremely vast amounts of memory and computational power, NVIDIA extended the Tensor Cores to support FP8, FP6, and FP4 [104, 105].

As CUDA and cuDNN are general-purpose platforms and libraries, NVIDIA provides engineers with TensorRT, a closed-source accelerated deep learning framework for inference on discrete GPUs or Jetson platforms [113, 114]. This allows developers to deploy the models efficiently and utilize APIs for quantization and sparsification [115]. To boost the adoption of LLMs onto their platforms, they additionally developed TensorRT-LLM [116], which provides optimized kernels for both floating-precision and quantized inference. TensorRT-LLM is the successor to the FasterTransformer [117] framework but it does not support the ViTs.

On top of CUDA and cuDNN, many deep learning frameworks have been built with early libraries like Lua Torch [118], Theano [119], and Caffe [120] to more modern frameworks like TensorFlow [121], PyTorch [122], and JAX [123]. TensorFlow and JAX are open-source frameworks developed by Google that integrate the XLA compiler [124] that translates Python [125] code written with those libraries into executable code on GPUs and TPUs. PyTorch is a modern version of Lua Torch governed by the Linux Foundation. PyTorch was designed with ease of use and targeted research development, with a current focus on research and the productization of deep learning models.

However, as those frameworks are general purpose, some are built to be an alternative with runtime-only lightweight targeting LLMs such as ggml [126] and vLLM [127]. Furthermore, Apache TVM [128] is a compiler stack for deep learning models that allows the conversion of the models developed from higher-level frameworks like PyTorch or TensorFlow to optimized device code. Moreover, TVM provides multiple intermediate representations (IR) to define the model’s layers and operations. As a result, it also provides options to find optimal configurations to execute kernels on target devices with search algorithms.

MLX [129] is an open-source framework from Apple targeting Apple Silicon to utilize unified memory architecture between ARM CPU and Metal GPU. Compared to PyTorch Metal backend, it can seamlessly select the most optimal execution device between those two modules. Unfortunately, to utilize Neural Engine for inference, the CoreML framework is required.

Most deep learning frameworks consider deep learning models to be graphs, and to unify those, the Multi-Level Intermediate Representation (MLIR) was created [130]. MLIR is a part of the LLVM compiler toolchain to unify the representation for the deep learning frameworks to streamline bringing new hardware for them. The power of MLIR is used by XLA (TensorFlow, PyTorch, and Jax), Triton [131] - Python domain-specific language (DSL) and compiler for NVIDIA GPUs, and Mojo [132]. Mojo is a language that is a superset of Python. It extends Python by adding static type checking and compilation to heterogeneous hardware while utilizing the Python ecosystem.

2.7 Conclusions

In this chapter, we presented the foundations of deep learning in the context of computer vision. We introduced the two fundamental neural network architectures - convolutional neural networks and vision transformers. Followed by an explanation of quantization. In the state-of-the-art review, we have explored all compression methods focusing on quantization in natural language processing and computer vision. All remaining chapters build on the work introduced here.

Chapter 3

Challenges in Post-Training Quantization of Vision Transformers

3.1 Introduction

Although the number of ViTs parameters is lower than that of LLMs there is still a need to introduce approximation methods such as quantization to reduce the computational, memory, and energy consumption of artificial intelligence systems [10]. The quantized DL models can be deployed in data center accelerators and at the edge. In both environments, quantized deep neural networks would consume less energy, and in the case of the data center, it could handle many more user requests simultaneously. Unfortunately, as mentioned in Chapter 2, the quantization techniques applied to convolutional neural networks do not typically transfer to ViT models.

In this chapter, we explore the limitations and challenges of post-training quantization methods applied to ViT models. Moreover, we provide a simple yet effective approach to quantizing the models with mixed precision. The contributions of this chapter are as follows:

- We evaluate the effect of post-training quantization on state-of-the-art ViT models: ViT, DeiT, Swin, and DeiT3.

- We hypothesize that the regularization applied during training positively affects quantization.
- Based on the calculation of the signal-to-quantization-noise ratio, we evaluate the sensitivity of the ViT layers to quantization.
- We fit linear regression between SQNR and quantization error for ViT family models and show a correlation between these two variables.
- We propose a mixed-precision approach for post-training quantization of pre-trained ViTs on the ImageNet1K dataset. We validate our approach and show that we can quantize up to 90% of all layers in the ViTs with negligible loss of accuracy.

3.2 Background

With the introduction of the ViT architecture [8], a plethora of variants and training recipes based on it have emerged [36, 40, 41, 37]. Dosovitskiy et al., [38] in their work, explored the self-supervised pre-training of transformers on JFT300M [133] and ImageNet21K [134]. They achieved higher performance on downstream tasks with a larger quantity of data and longer pre-training.

While the ViT performance scales with the amount of data and compute hours, it might not be feasible in constrained resource environments. DeiT [36] was trained on ImageNet1K only [22] with a knowledge distillation framework. Moreover, they show that training the ViT model with additional data augmentation and regularization techniques is possible without a large pre-training step.

In addition, Steiner et al. conducted an extensive study to examine the effects of data augmentation, regularization, patch size, pre-training, optimizer, and other training variables to determine the best recipe for training ViT [135]. Ultimately, they trained over fifty thousand models on the ImageNet21K dataset to match or outperform the corresponding models on the JFT300M dataset.

However, to address the shortcomings of the ViTs, Liu et al. proposed the Swin Transformer, which introduces hierarchical feature maps and shifted window attention [40]. Both features improve model performance for downstream tasks such as image classification or instance segmentation. In addition, the authors apply a data augmentation regime similar to that of the authors of the DeiT architecture [36].

Furthermore, during the training of the deeper ViT models, it was found that the model may not converge optimally. To remedy this, LayerScale was proposed by Touvron et al. [42] and a modified version was added to the DeiT3 architecture [41].

Additional parameters added to the residual connection were shown to stabilize convergence and improve model performance. Moreover, in DeiT3, the authors show that it is possible to train the model with a simple data augmentation regime but with a modified training recipe - training the models on the smaller image sizes while gradually increasing the image size.

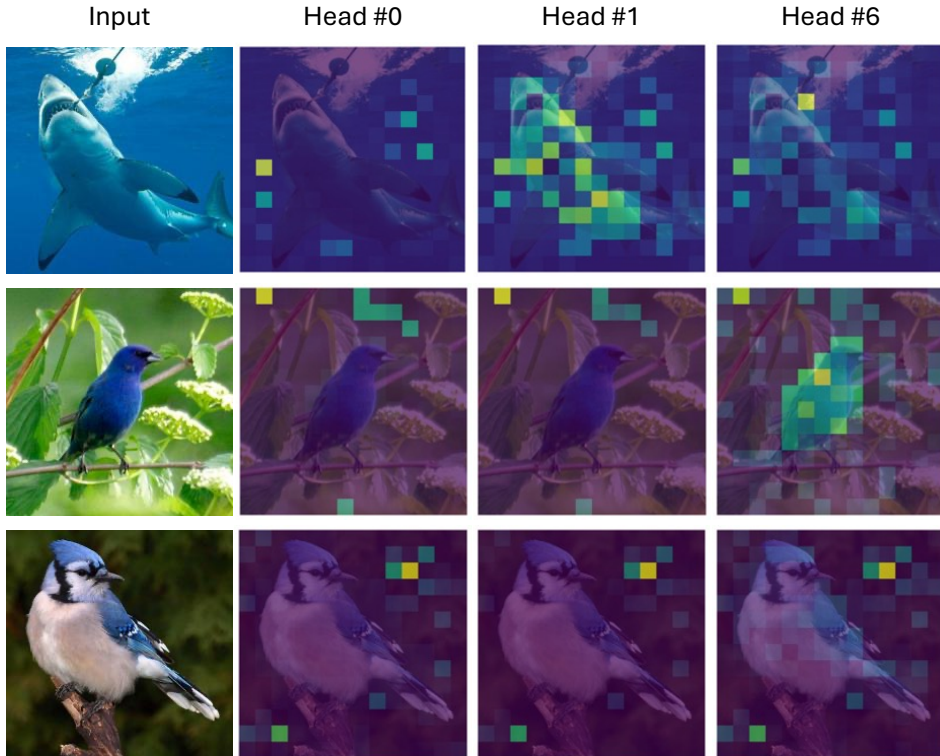


Figure 3.1: This image displays attention heatmaps created by the final self-attention module of a ViT-B/16/224 model using sample ImageNet1K validation images¹. The color intensity corresponds to the softmax activation values, revealing the model’s attention patterns and indicating the most important regions of the image for classification. Interestingly, some of the model’s attention focuses on the background rather than the object itself.

¹The figure was created by inferring the input image with a pre-trained ViT-B/16/224 model. The model’s weights were fetched from the timm package. At heads 0, 1, and 6, we attached forward hooks to the softmax operation in the attention head. Then, activation maps were upsampled to match the input image size.

The issues arising during the training of the models from the ViTs family make them ideal candidates for applying compression methods like post-training quantization. However, the quantization might clip the activation maps due to ViTs’ building blocks such as the LayerNorm and attention module. Moreover, recent work has shown that attention modules in CV utilize the background as a scratchpad [136, 137]. As a result, the attention head that is supposed to ignore the current output produces an outlier in the activation map, as shown in Figure 3.1. Two methods were proposed to change the ViT architecture to mitigate outliers. One proposed a gate that thresholds the activations to teach the attention heads not to produce outliers. While the other added a register mechanism to the model’s architecture to let the ViT model write to them. The register tokens are concatenated to the input patches. As a result, the model can write the computations and store the outliers in those patches. However, those register tokens are not used for downstream tasks, e.g., classification task.

In this chapter, we present an exhaustive experimental study on the post-training quantization of ViT, DeiT, Swin Transformer (Swin), and DeiT3 architectures.

3.3 Methodology

For our experimental study, we focused on static and dynamic post-training quantization; see Figure 2.8. We utilized the PyTorch native quantization framework to conduct our experiments with simulated quantization. Nonetheless, we are able to accelerate computations on GPU by introducing quantization and dequantization functions (as defined in Equations (2.10), (2.12) and (2.14)) before and after each operation, while the operation itself is performed in FP arithmetic. As a result we obtain the integer numerical equivalent of quantized operators.

We converted the model’s weights to INT8 per-channel symmetric quantization using a min-max observer. The quantization parameters for static quantization were estimated based on the 2000 random samples from the ImageNet1K training dataset. For LayerNorm, GELU, and attention, we used scalar asymmetric UINT8 quantization with a min-max observer. The pre-trained vision transformers ViT, DeiT, Swin, and DeiT3 checkpoints were fetched from the *timm* library [138]. The following features represent all variants of these models. Moreover, these parameters affect the model’s top-1 accuracy, sequence length, and total multiply-add operation (MAC). In our setup, we considered the following model features:

- size of the transformer: tiny, small, base, large, and huge (DeiT3 only),
- patch size: 4 (Swin only), 8 (ViT only), 16, and 32,
- input image size: 224 and 384,

- knowledge distillation for DeiT.

We compared the statically and dynamically quantized models with the baseline FP32 model’s top-1 accuracy on the ImageNet1K validation set. Moreover, we evaluated it against PTQ4ViT [98], reproduced the results of FQ-ViT [96], and extended the FQ-ViT models’ evaluation to the DeiT3 architecture.

We recorded the output activation distribution of the fully connected layers of the FP ViT, DeiT, and DeiT3 models on 2000 random samples from the ImageNet1K training set. These models share the architectural blocks that allowed us to compare the activation distributions. For the analysis, we selected the models with a patch size $P = 16$ and an input image size of 224×224 . We also evaluated the sensitivity of the statically quantized ViT, DeiT, and DeiT3 models to their reference FP models using the SQNR metric (Equation (2.13)) for weights and activations. Activation measurements were computed using over 2000 random samples from the ImageNet1K validation set. We also fitted the linear regression to the relationship between the average SQNR value of the fully connected layers and the quantization error.

Having established the baseline quantization performance of the quantized DL models and sensitivity analysis, we systematically searched partial quantization nodes on the ViT models that would result in low quantization error. We started from the baseline models with their quantized performance presented in Table 3.1 and Table 3.2. Each node’s DL models are considered or not for quantization with equal probability. We kept the skip connection in FP arithmetic. We evaluated the model’s performance at various quantization thresholds, starting with 50% nodes quantized with steps of 5% up to 95%. We ran the experiments ten times for random mixed-precision quantization and reported the mean and standard deviation of the quantization error on the ImageNet1K validation dataset. Lastly, we assessed the memory storage savings from the mixed-precision quantization compared to the FP ViT, DeiT, DeiT3, and Swin Transformer models.

3.4 Results

In this section, we present our empirical results of applying post-training quantization on ViT, DeiT, DeiT3, and Swin Transformer models. First, we establish the baseline quantization in Section 3.4.1. Next, in Section 3.4.2, we present an analysis of the activation distribution of MLP and attention fully connected layers of ViT, DeiT, and DeiT3. Moreover, we assess quantization sensitivity analysis of weights and activations. Finally, in Section 3.4.3, we propose a mixed-precision quantization scheme for applying post-training quantization to ViT, DeiT, DeiT3, and Swin Transformer.

3.4.1 Post-training Quantization of Vision Transformers

We first observed that the ViT architecture is sensitive to INT8 static quantization, with a significant drop in the top-1 accuracy of the ViT model. With INT8 static quantization, we expected the drop in model accuracy to be 1-2% similarly to, CNNs such as ResNet which showed a minimal drop in top-1 accuracy with INT8 static quantization [12]. In addition, as shown in Table 3.1, dynamic quantization preserved the ViT model’s predictive power for most models, achieving higher top-1 accuracy than INT8 static quantization and FQ-ViT.

For ViT-T/16/224, ViT-T/16/384, and ViT-B/32/224 the top-1 accuracy of INT8 dynamic quantization (INT8-D) dropped by more than 2%. With our reproduced FQ-ViT results, we found that except for ViT-T/16/224, ViT-T/16/384, ViT-S/32/224, ViT-S/32/384, and ViT-B/32/224, the performance of the ViT models were similarly to dynamic quantization within 2% top-1 accuracy degradation range. Nevertheless, for all the ViT models, dynamic quantization achieved higher top-1 accuracy. This could be explained by the fact that dynamic quantization was only applied to fully connected layers within the MLP block, and the rest of the operations, including the attention mechanisms, were performed in FP32. However, PTQ4ViT, which used Hessian-guided calibration, outperformed all other methods for the models reported by the authors.

Interestingly, the robustness of the ViT model to static quantization did not increase with model size. Unfortunately, the overparameterization of the model did not help to recover the predictive power. Since dynamic quantization did not significantly affect the top-1 accuracy of the models, we should also investigate in the future the activation maps of other operations.

Contrary to ViTs, Swin Transformer models showed robustness to the static quantization. These models lost, on average $5.17\% \pm 3.93\%$ top-1 accuracy, with the smallest drop in performance experienced in Swin-T/4/224 and the largest in Swin-B/4/224. Nevertheless, similar to ViTs, the best performance was obtained by applying dynamic quantization or PTQ4ViT. FQ-ViT did improve the predictive power over the static quantization, especially for Swin-L/4/224, where the top-1 accuracy degradation dropped to 1.32% with respect to the baseline FP32 model. At the same time, PTQ4ViT achieved 85.14% top-1 accuracy for Swin-B/4/224, compared to 74.35% and 84.99% for static and dynamic quantization, respectively, for the same model.

In Table 3.2, we present the effect of the post-training quantization on the DeiT and DeiT3 architectures. We observed that the distilled versions of the DeiT models were more robust to quantization than the non-distilled ones. On average, non-distilled versions lost $2.24\% \pm 1.41\%$ compared to $0.65\% \pm 0.66\%$. Moreover, even though the DeiT shares the same architecture as ViT, it was more

Table 3.1: Top-1 accuracy of post-training quantization on the ImageNet1K validation set for ViT and Swin Transformer models. We explored static (INT8) and dynamic quantization (INT8-D). The results are compared to FQ-ViT and PTQ4ViT. Model labels specify model name/patch size/input size. The best quantized top-1 score is highlighted in bold.

Model	FP32	INT8	INT8-D	FQ-ViT	PTQ4ViT
ViT-T/16/224	75.45	0.24	70.94	45.80	-
ViT-T/16/384	78.44	0.12	74.45	45.18	-
ViT-S/16/224	81.38	55.01	79.27	78.58	81.0
ViT-S/16/384	83.80	42.52	82.36	81.61	-
ViT-S/32/224	75.99	12.46	73.95	59.13	75.58
ViT-S/32/384	80.48	28.85	78.70	68.43	-
ViT-B/8/224	86.23	70.61	85.65	83.70	-
ViT-B/16/224	85.10	65.48	83.96	83.70	84.25
ViT-B/16/384	85.99	30.00	85.03	84.11	85.82
ViT-B/32/224	80.73	26.67	78.38	70.99	-
ViT-B/32/384	83.35	69.21	82.64	81.37	-
ViT-L/16/224	85.84	12.88	85.29	84.97	-
ViT-L/16/384	87.10	8.56	86.59	81.36	-
Swin-T/4/224	81.38	79.02	80.84	80.02	81.24
Swin-S/4/224	83.32	80.25	83.20	82.40	83.10
Swin-B/4/224	85.28	74.35	84.99	82.50	85.14
Swin-L/4/224	86.32	81.99	86.15	85.61	-

robust to static quantization. While dynamic quantization achieved higher top-1 accuracy for DeiT-T/16/224 and DeiT-T/16/224/D, static quantization did not incur much damage to the predictive power of those models to not consider it. Moreover, FQ-ViT reached greater top-1 accuracy than static quantization in the case of DeiT-T/16/224, DeiT-S/16/224, and DeiT-B/16/224. Finally, the PTQ4ViT algorithm outperformed the FQ-ViT and INT8 static and dynamic quantization for DeiT-S/16/224 and DeiT-B/16/224.

Finally, the DeiT3 architecture’s the best top-1 accuracy was obtained by applying dynamic quantization. Unfortunately, the static quantization introduced high quantization errors for the DeiT3-B/16/224 and larger models. Similarly, the FQ-ViT algorithm failed to quantize the models larger than DeiT3-M/16/224.

3. CHALLENGES IN POST-TRAINING QUANTIZATION OF VISION TRANSFORMERS

Table 3.2: Top-1 accuracy of post-training quantization on the ImageNet1K validation set for DeiT and DeiT3 models. We report static (INT8) and dynamic quantization (INT8-D). The results are compared to FQ-ViT and PTQ4ViT. Model labels include model name/patch size/input size, and whether the model was distilled. The best quantized top-1 score is highlighted in bold.

Model	FP32	INT8	INT8-D	FQ-ViT	PTQ4ViT
DeiT-T/16/224	72.16	71.37	71.56	71.05	-
DeiT-T/16/224/D	74.52	73.85	73.99	73.63	-
DeiT-S/16/224	79.85	77.51	78.91	78.49	79.47
DeiT-S/16/224/D	81.22	80.64	80.64	80.42	-
DeiT-B/16/224	81.98	78.38	81.29	80.96	81.48
DeiT-B/16/224/D	83.39	82.68	82.44	82.48	-
DeiT3-S/16/224	81.36	76.89	80.33	79.62	-
DeiT3-S/16/384	83.43	52.18	82.62	81.28	-
DeiT3-M/16/224	83.09	72.34	82.53	82.1	-
DeiT3-B/16/224	83.78	4.32	83.30	0.10	-
DeiT3-B/16/384	85.08	2.50	84.71	0.10	-
DeiT3-L/16/224	84.78	0.15	84.62	0.10	-
DeiT3-L/16/384	85.82	0.15	85.73	0.10	-
DeiT3-H/14/224	85.22	29.97	85.05	0.10	-

Within our experiments, we have observed that DeiT and Swin Transformer architectures are more robust to the quantization of the ViT and DeiT3. The common denominator between DeiT and Swin Transformer is the data augmentation applied during training. Moreover, the distillation from the CNN increased the robustness of the DeiT model. Meanwhile, the ViT and DeiT3 had simple or no data augmentation during training. We hypothesize that the regularization applied during training positively affects the robustness of the quantization of the ViT family models.

3.4.2 Activation Distribution and Sensitivity Analysis

The first observation in the FP ViT, DeiT, and DeiT3 models was that each fully connected layer produced outliers across the different model sizes. For the presentation, we have selected the third (zero-indexed) and penultimate MLP blocks and the penultimate attention projection fully connected layer. In Figure 3.2

and Figure 3.3, we present the activation distribution of the first and second fully connected layers of the MLP block in the second block - *blocks.2.mlp.fc1* and *blocks.2.mlp.fc2*.

Within models, we noticed that the median output activation range moved from negative to zero. This could be explained by the GELU activation function between those fully connected layers that gated the negative activation range. Based on these plots, we observed that in the ViT family, the outlier activation range is larger in the first fully connected layer within this block than in the second fully connected layer, except for ViT-S/16/224.

Next, as expected within the DeiT family, we observed that the distilled version had fewer outlier activations, especially in the second fully connected layer as shown in Figure 3.3. Except for the DeiT-B/16/224/D, we observed more negative outliers in the activation maps in the first fully connected layer. We also found that the ViT activation ranges were more extensive than those of the DeiT.

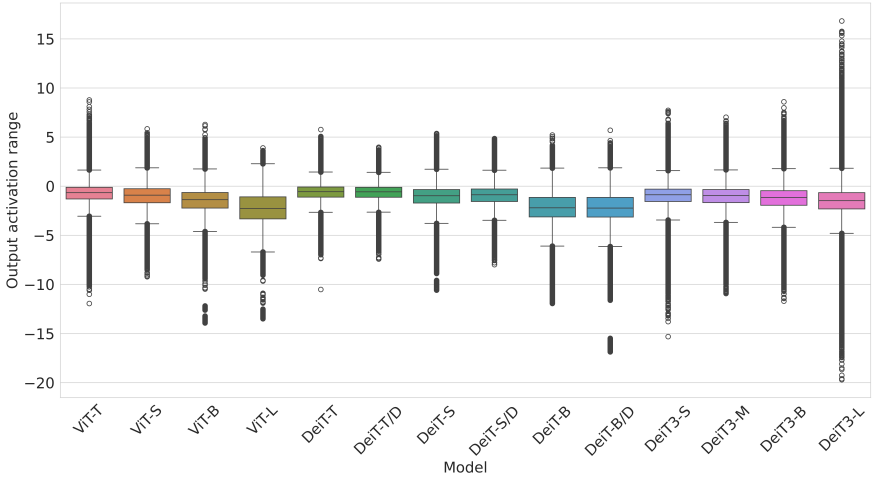


Figure 3.2: This box plot shows the output activation distribution of the *blocks.2.mlp.fc1* fully connected layer in ViT, DeiT, and DeiT3. The input image size for each model is 224×224 , and the patch size is $P = 16$. The points outside the whiskers of the box plot represent outliers in the output activations.

In addition, the DeiT3 models exhibited even more considerable output activation outlier ranges. Changes in the DeiT3 architecture include the addition of the LayerScale layer and reduced augmentation applied during training. Even in the second fully connected layer, the maximum values were larger, and the minimum

3. CHALLENGES IN POST-TRAINING QUANTIZATION OF VISION TRANSFORMERS

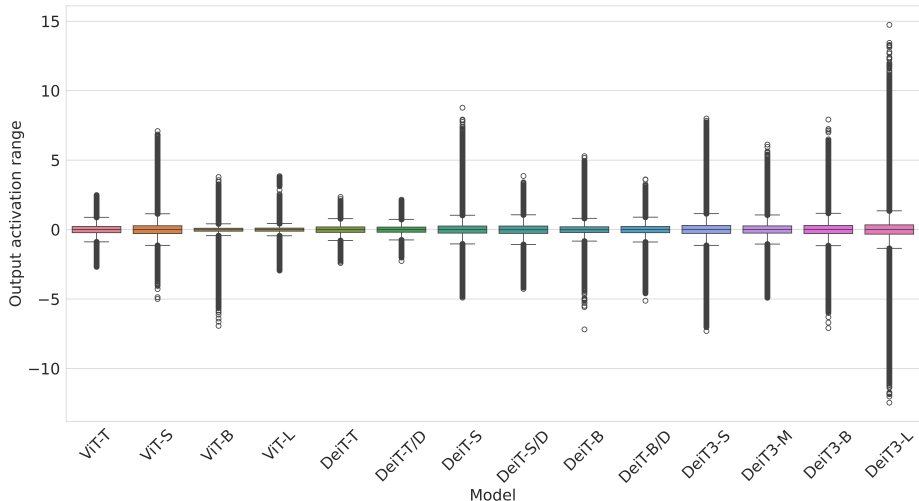


Figure 3.3: This box plot shows the output activation distribution of the *blocks.2.mlp.fc2* fully connected layer in ViT, DeiT, and DeiT3. The input image size for each model is 224×224 , and the patch size is $P = 16$. The points outside the whiskers of the box plot represent outliers in the output activations.

values were smaller than those in the equivalent ViT model. These activation ranges across ViT and DeiT3, which could explain why these models lost most of their predictive power after INT8 static quantization. As shown in Table 2.5, the FP32 data type has a broader representation format than INT8 and UINT8. If, during the calibration phase, these outliers were not recorded, they would be clipped to a minimum and maximum value of the INT8 or UINT8. Moreover, if these outliers were recorded, the quantization scale factor would impact the values closer to the median and could be rounded to the improper nearest integer. Moreover, the DeiT3-L/16/224 outlier activation range is the largest across all the models in the MLP layer in the second block.

The fully connected QKV layer in the attention block is crucial to compute the query, key, and value weights for attention. Figure 3.4 shows the activation ranges in the tenth attention block (*blocks.10.attn.qkv*). Within these ranges, we observed that the median is zero for all models. Furthermore, for most models, the minimum and maximum values are almost symmetrical with respect to the median. Likewise in previous fully connected layers, we registered the outliers in all models. Surprisingly, the outliers in DeiT-B/16/224/D were more prominent

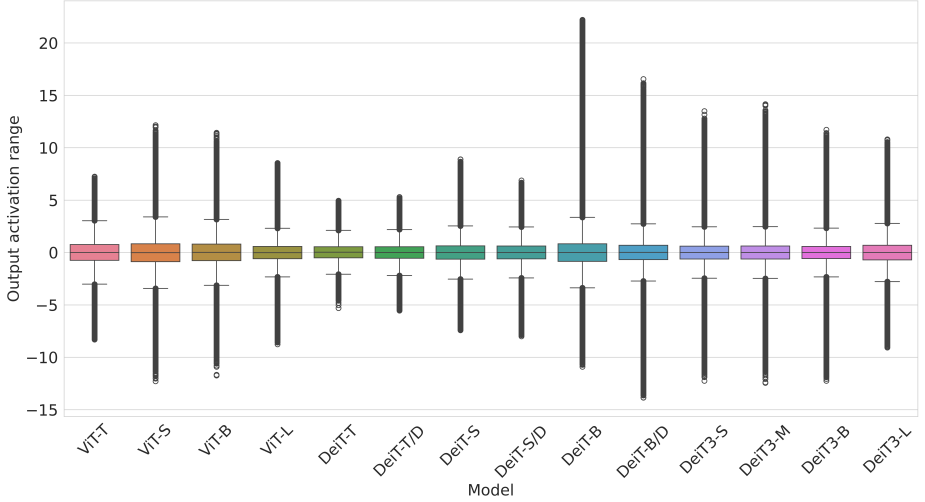


Figure 3.4: This box plot shows the output activation distribution of the *blocks.10.attn.qkv* fully connected layer in ViT, DeiT, and DeiT3. The input image size for each model is 224×224 , and the patch size is $P = 16$. The points outside the whiskers of the box plot represent outliers in the output activations.

than those of ViT-B/16/224. Even with the outliers recorded at the output in the tenth QKV fully connected layer, the SQNR value for DeiT-B/16/224 and DeiT-B/16/224/D was higher than that of ViT-B/16/224 and DeiT-B/16/224, as shown in Figure 3.7.

Furthermore, in the first fully connected layer of the tenth MLP block, we observed a similar activation range for most models compared to the corresponding layer of the second MLP block, i.e., the median activation range is below zero. However, as shown in Figure 3.5, the broadest ranges were recorded in ViT-S/16/224, ViT-B/16/224, and DeiT-B/16/224. The outliers in ViT-S/16/224 and ViT-B/16/224 were negatively skewed, while they were positively skewed for DeiT-B/16/224. Interestingly, we observed here the effect of knowledge distillation on the DeiT architecture, as DeiT-T/16/224/D, DeiT-S/16/224/D, and DeiT-B/16/224/D recorded lower activation ranges and fewer outliers compared to non-distilled variants. Surprisingly, DeiT3-L/16/224 compared to small, medium, and base DeiT3 and ViT-L/16/224 also showed a smaller activation range in this layer.

In addition, we recorded many outliers in the second fully connected layer in the tenth MLP block, ViT-S/16/224, ViT-B/16/224, and DeiT-B/16/224, as shown

3. CHALLENGES IN POST-TRAINING QUANTIZATION OF VISION TRANSFORMERS

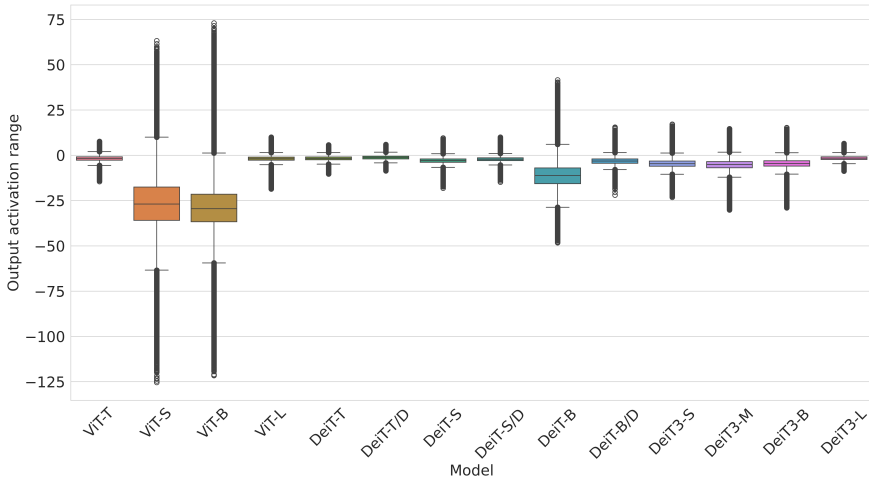


Figure 3.5: This box plot shows the output activation distribution of the *blocks.10.mlp.fc1* fully connected layer in ViT, DeiT, and DeiT3. The input image size for each model is 224×224 , and the patch size is $P = 16$. The points outside the whiskers of the box plot represent outliers in the output activations.

in Figure 3.6. However, the activation range between the recorded minimum and maximum values was smaller than in the first fully connected layer. Interestingly, within ViT-L/16/224, DeiT-S/16/224, and DeiT-B/16/224, we observed increased maximum absolute values in registered outliers. Furthermore, similar to the previous second fully connected layer in the second MLP block, the median value for the activation ranges in the ViT, DeiT, and DeiT3 models was equal to zero.

After analyzing the activation ranges in the FP ViT, DeiT, and DeiT3 models, it was expected that a gradual decrease in signal flow would occur through the activations of these models. Initially, in ViT-B/16/224, we observed that the SQNR dropped sharply below zero, as shown in Figure 3.7. Although the signal recovered at the classification level (head) in the later stages, it dropped below zero again previously multiple times. Within ViT-S/16/224, the activation signal did not drop as drastically as in ViT-B/16/224. Nevertheless, the signal vanished at *blocks.10.mlp.fc2* layer, in which we previously showed a considerable range of outliers. In these models, we observed a recurring pattern where the first fully connected layer in the MLP block had a higher SQNR value than the second fully connected layer in the same block.

Next, we observed more pronounced signals within activations across all layers in DeiT-S/16/224/D and DeiT-B/16/224/D than in ViT-S/16/224 and ViT-B/16/224.

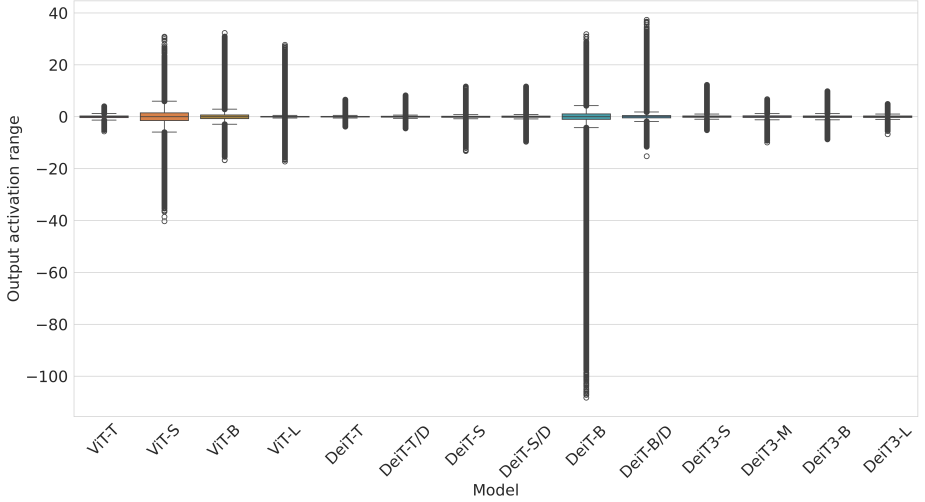


Figure 3.6: This box plot shows the output activation distribution of the *blocks.10.mlp.fc2* fully connected layer in ViT, DeiT, and DeiT3. The input image size for each model is 224×224 , and the patch size is $P = 16$. The points outside the whiskers of the box plot represent outliers in the output activations.

In addition, the signal was still present in the classification head of the models. Especially after *blocks.6.mlp.fc1*, the signal improved in these models, while the ViT decreased towards the last layer.

Finally, within DeiT3-S/16/224 and DeiT3-B/16/224, we observed the SQNR values between ViT and DeiT models. Similarly, we observed higher SQNR values in the first than in the second fully connected layers in the MLP block. Interestingly, the DeiT3-S/16/224 head classification layer had a higher SQNR value than the DeiT-S/16/224/D. However, the DeiT3-B/16/224 SQNR value dropped below zero in the last and eighth fully connected layers of the MLP block.

Moreover, in Figure 3.8, we plotted the weight SQNR between the INT8 and FP models. We first noticed that the SQNR was prominent in all the models' weights. Similarly, the SQNR weight value decreased in the second fully connected layer of the MLP block compared to the first for activations. From these SQNR weight values, we could conclude that the ViT family models are robust to dynamic quantization. In comparison, activations within SQNR showed that the ViT family models are susceptible to activation quantization.

Lastly, we fitted the linear regression between the average model SQNR value of the fully connected layers and the quantization error, as shown in Figure 3.8.

3. CHALLENGES IN POST-TRAINING QUANTIZATION OF VISION TRANSFORMERS

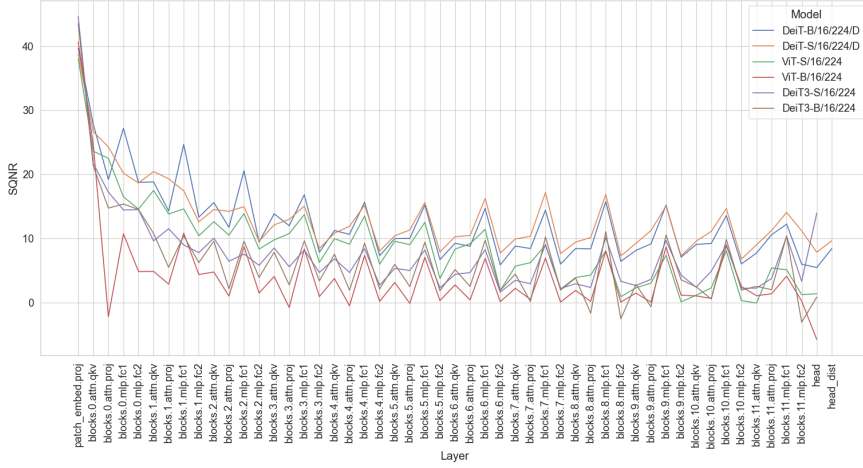


Figure 3.7: SQNR Ratio Analysis of ViT Models: This figure compares SQNR values for small and base ViT family models. SQNR of activations measured after each layer, illustrating the impact of quantization on intermediate representations.

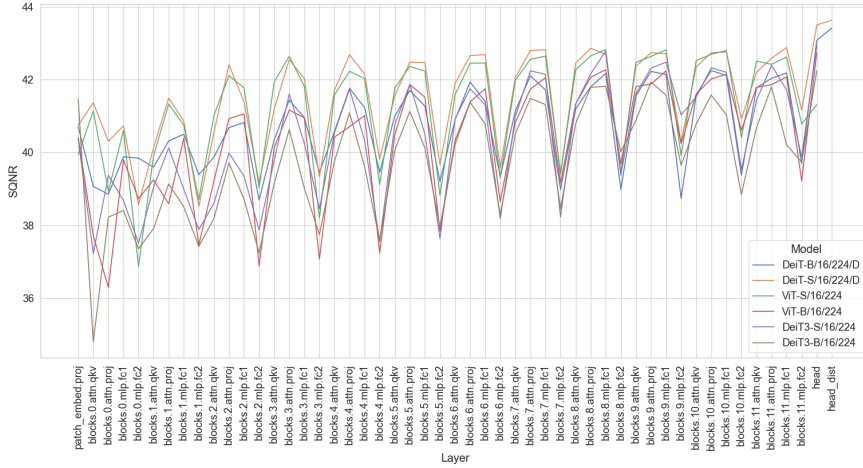


Figure 3.8: SQNR Ratio Analysis of ViT Models: This figure compares SQNR values for small and base ViT family models. SQNR values between FP32 model weights and INT8 quantized weights.

We observed a negative correlation with $\rho = -0.36$, 90% CI $[-0.67, 0.045]$ that confirmed that we expect the quantization error to decrease for higher SQNR values. Furthermore, we noticed that the model’s families created clusters in the plot. With a couple of outliers like DeiT3-S/16/224, the average SQNR value was 7.60, with a quantization error equal to 4.47%. Nevertheless, the highest average SQNR value was observed in the DeiT family, which reported the smallest quantization error.

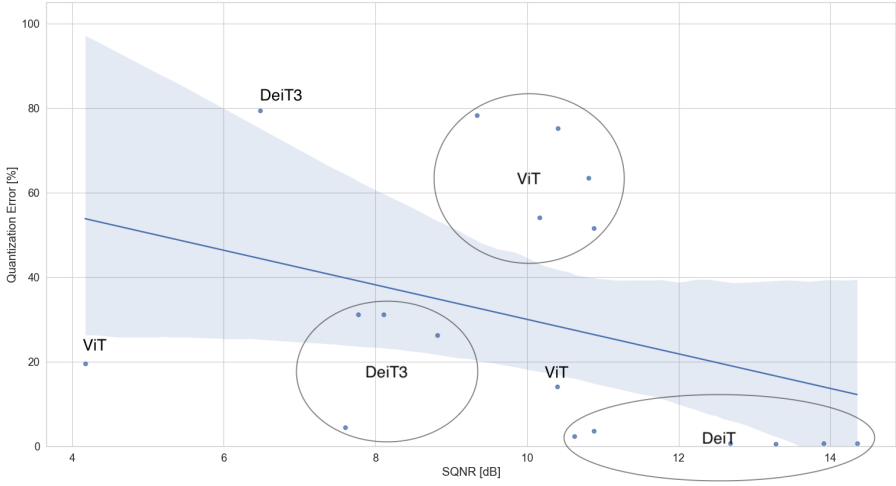


Figure 3.9: Linear regression between average SQNR value of fully connected layers and quantization error of ViT, DeiT, and DeiT3. We observed a negative correlation between SQNR and quantization error with $\rho = -0.36$, 90% CI $[-0.67, 0.045]$. Circles indicate models belonging to one of the model families.

3.4.3 Mixed-precision Quantization of Vision Transformers

We evaluated the robustness of the ViT model to mixed-precision quantization at various quantized operation nodes (abbreviated as Op.). Starting from 50% up to 95% of the operations were quantized. As a result, the percentage of the node’s arithmetic was considered to be performed in INT8, while the rest was computed in FP32. We did not consider the add operation of the skip connection layer for quantization, as it helps the signal to pass through the network for deeper layers.

3. CHALLENGES IN POST-TRAINING QUANTIZATION OF VISION TRANSFORMERS

Table 3.3: Partial mixed-precision quantization results on ImageNet1K: This table presents the top-1 quantization error (mean $\pm$ standard deviation) for ViT, DeiT, and Swin tiny models on the ImageNet1K validation set. Results are averaged over ten runs. The lowest quantization error with the highest percentage of operations (Op.) quantized is highlighted in bold for each model.

Op. %	ViT-T/16/224	ViT-T/16/384	DeiT-T/16/224	DeiT-T/16/224/D	Swin-T/4/224
50%	29.71 $\pm$ 21.94	32.38 $\pm$ 14.09	0.29 $\pm$ 0.08	0.20 $\pm$ 0.05	1.11 $\pm$ 0.37
55%	22.98 $\pm$ 9.44	24.53 $\pm$ 9.70	0.33 $\pm$ 0.07	0.27 $\pm$ 0.08	1.07 $\pm$ 0.45
60%	41.39 $\pm$ 15.30	45.22 $\pm$ 15.78	0.37 $\pm$ 0.05	0.33 $\pm$ 0.07	1.22 $\pm$ 0.34
65%	41.78 $\pm$ 12.22	45.52 $\pm$ 12.36	0.39 $\pm$ 0.04	0.31 $\pm$ 0.10	1.27 $\pm$ 0.49
70%	47.25 $\pm$ 13.76	51.20 $\pm$ 14.24	0.40 $\pm$ 0.09	0.32 $\pm$ 0.08	1.10 $\pm$ 0.33
75%	46.80 $\pm$ 15.16	51.49 $\pm$ 15.70	0.41 $\pm$ 0.06	0.36 $\pm$ 0.07	1.37 $\pm$ 0.41
80%	54.72 $\pm$ 6.49	59.11 $\pm$ 5.94	0.41 $\pm$ 0.07	0.40 $\pm$ 0.06	1.57 $\pm$ 0.41
85%	56.29 $\pm$ 9.58	60.86 $\pm$ 9.43	0.47 $\pm$ 0.07	0.39 $\pm$ 0.07	1.61 $\pm$ 0.27
90%	62.22 $\pm$ 10.07	66.69 $\pm$ 9.76	0.50 $\pm$ 0.04	0.41 $\pm$ 0.06	1.70 $\pm$ 0.30
95%	66.04 $\pm$ 2.02	69.50 $\pm$ 1.91	0.46 $\pm$ 0.04	0.39 $\pm$ 0.04	2.27 $\pm$ 0.07

Even with the mixed-precision partial quantization of the ViT-T models, the quantization error of the model was too large to deliver a stable model. Since the model parameters take up 27MB of memory in FP32, we recommend keeping it at that precision or FP16. Nevertheless, using our method, the DeiT-T/16/224 reduced the quantization error to $0.50\% \pm 0.04\%$ and $0.46\% \pm 0.04\%$ for 90% and 95% of the quantized operations, as observed in Table 3.3. As a result, the size of the quantized model was reduced to 6.58 MB. However, for the distilled version of DeiT-T/16/224/D, we did not observe significant improvements in top-1 accuracy between the baseline post-training quantization and the mixed-precision quantization. Furthermore, for Swin-T/4/224, we were able to quantize up to 90% of the operation nodes where the model outperformed the baseline INT8 and FQ-ViT quantization methods. With our method, Swin-T/4/224 required up to 37.05 MB of memory to hold the weights, which is 10 MB more than ViT-T/16/224, but with 4.86% higher top-1 accuracy.

With ViT-S/16/224, we reduced the quantization error from 26.37% to $3.21\% \pm 0.45\%$ at 80% quantized nodes, see Table 3.4, with the best result outperforming the FQ-ViT algorithm. The mixed-precision quantized model required up to 35.32 MB of memory. As a result, it was smaller than the mixed-precision quantized Swin-T/4/224 with comparable top-1 accuracy. Similarly, the quantization error of ViT-S/32/224 dropped from 63.53% to 5.75% for a rate of 70% quantized nodes. Nevertheless, the quantization error was too high for the model to be useful at this quantization level. DeiT-S/16/224/D achieved half the quantization error compared to the baseline INT8 static quantization for 90% of the quantized nodes.

Table 3.4: Partial mixed-precision quantization results on ImageNet1K: This table presents the top-1 quantization error (mean $\pm$ standard deviation) for ViT, DeiT, DeiT3, and Swin small and base models on the ImageNet1K validation set. Results are averaged over ten runs. The lowest quantization error with the highest percentage of operations (Op.) quantized is highlighted in bold for each model.

Op. %	ViT-S/16/224	ViT-S/32/224	DeiT-S/16/224	DeiT-S/16/224/D	DeiT3-S/16/224	DeiT3-M/16/224	Swin-S/4/224
50%	1.82 $\pm$ 0.79	14.77 $\pm$ 8.44	0.52 $\pm$ 0.12	0.19 $\pm$ 0.08	0.74 $\pm$ 0.20	2.37 $\pm$ 0.72	0.27 $\pm$ 0.10
55%	2.12 $\pm$ 0.48	11.25 $\pm$ 6.61	0.55 $\pm$ 0.15	0.22 $\pm$ 0.07	0.96 $\pm$ 0.18	3.38 $\pm$ 2.01	0.34 $\pm$ 0.04
60%	2.43 $\pm$ 0.67	18.09 $\pm$ 6.39	0.71 $\pm$ 0.15	0.27 $\pm$ 0.05	0.92 $\pm$ 0.23	3.06 $\pm$ 0.55	0.33 $\pm$ 0.09
65%	2.42 $\pm$ 0.50	17.47 $\pm$ 7.36	0.71 $\pm$ 0.10	0.23 $\pm$ 0.08	1.09 $\pm$ 0.14	3.69 $\pm$ 1.36	0.38 $\pm$ 0.07
70%	2.73 $\pm$ 0.61	20.73 $\pm$ 7.65	0.78 $\pm$ 0.14	0.25 $\pm$ 0.06	1.17 $\pm$ 0.30	3.53 $\pm$ 1.20	0.38 $\pm$ 0.08
75%	2.75 $\pm$ 0.81	16.53 $\pm$ 6.35	0.79 $\pm$ 0.14	0.27 $\pm$ 0.06	1.25 $\pm$ 0.25	3.54 $\pm$ 0.55	0.42 $\pm$ 0.03
80%	3.21 $\pm$ 0.45	21.04 $\pm$ 5.90	0.82 $\pm$ 0.21	0.28 $\pm$ 0.04	1.26 $\pm$ 0.21	3.51 $\pm$ 0.56	0.42 $\pm$ 0.06
85%	3.93 $\pm$ 0.31	26.74 $\pm$ 3.98	0.93 $\pm$ 0.06	0.30 $\pm$ 0.05	1.36 $\pm$ 0.17	4.09 $\pm$ 0.47	0.46 $\pm$ 0.05
90%	3.83 $\pm$ 0.53	27.37 $\pm$ 6.84	0.92 $\pm$ 0.12	0.28 $\pm$ 0.05	1.45 $\pm$ 0.10	3.90 $\pm$ 0.19	0.43 $\pm$ 0.06
95%	4.03 $\pm$ 0.53	29.88 $\pm$ 1.18	0.96 $\pm$ 0.07	0.41 $\pm$ 0.02	1.68 $\pm$ 0.10	3.84 $\pm$ 0.27	0.58 $\pm$ 0.05
Op. %	ViT-B/8/224	ViT-B/16/224	ViT-B/32/224	DeiT-B/16/224	DeiT-B/16/224/D	DeiT3-B/16/224	Swin-B/4/224
50%	1.86 $\pm$ 1.62	5.96 $\pm$ 5.43	4.21 $\pm$ 1.75	1.20 $\pm$ 0.24	0.26 $\pm$ 0.10	15.21 $\pm$ 18.97	0.50 $\pm$ 0.07
55%	1.74 $\pm$ 1.49	6.21 $\pm$ 5.72	3.55 $\pm$ 1.15	1.29 $\pm$ 0.22	0.31 $\pm$ 0.08	17.17 $\pm$ 17.24	0.63 $\pm$ 0.09
60%	2.35 $\pm$ 1.29	8.55 $\pm$ 5.21	5.35 $\pm$ 1.55	1.33 $\pm$ 0.20	0.35 $\pm$ 0.06	12.96 $\pm$ 15.14	0.57 $\pm$ 0.07
65%	2.26 $\pm$ 1.47	7.48 $\pm$ 5.44	5.06 $\pm$ 1.78	1.47 $\pm$ 0.23	0.35 $\pm$ 0.06	31.63 $\pm$ 30.15	0.61 $\pm$ 0.17
70%	3.11 $\pm$ 0.88	10.84 $\pm$ 3.45	5.69 $\pm$ 1.57	1.51 $\pm$ 0.29	0.38 $\pm$ 0.10	18.44 $\pm$ 23.42	0.75 $\pm$ 0.12
75%	2.88 $\pm$ 1.21	9.85 $\pm$ 4.65	5.43 $\pm$ 1.15	1.46 $\pm$ 0.21	0.38 $\pm$ 0.07	24.02 $\pm$ 19.30	0.83 $\pm$ 0.09
80%	2.54 $\pm$ 1.27	8.90 $\pm$ 5.36	6.48 $\pm$ 1.10	1.67 $\pm$ 0.09	0.43 $\pm$ 0.03	18.83 $\pm$ 23.35	0.81 $\pm$ 0.14
85%	3.32 $\pm$ 0.07	12.43 $\pm$ 0.23	7.43 $\pm$ 1.12	1.76 $\pm$ 0.11	0.41 $\pm$ 0.06	45.93 $\pm$ 22.05	0.92 $\pm$ 0.10
90%	3.18 $\pm$ 0.90	11.30 $\pm$ 3.56	7.96 $\pm$ 1.77	1.69 $\pm$ 0.06	0.42 $\pm$ 0.09	40.09 $\pm$ 19.83	1.01 $\pm$ 0.04
95%	3.35 $\pm$ 0.05	11.00 $\pm$ 0.09	8.75 $\pm$ 0.27	1.82 $\pm$ 0.08	0.45 $\pm$ 0.04	24.40 $\pm$ 22.60	1.00 $\pm$ 0.10

In addition, DeiT-S/16/224 improved top-1 accuracy over PTQ4ViT at 80% of quantized nodes and over INT8 static quantization at 90% quantization level. This resulted in the model requiring up to 40% and 32.5% of memory to store the weights compared to the FP32 model; see Figure 3.10. At 90% quantized nodes, the model approaches the size of FP32 ViT-T/16/224 while achieving higher top-1 accuracy.

Likewise to the static post-training quantization of DeiT3 models, we observed that they were more vulnerable to mixed-precision quantization than the DeiT architecture. With DeiT3-S/16/224, our method achieved a four times smaller quantization error than FQ-ViT at 70% of the quantized nodes. However, for DeiT3-M/16/224, we obtained the best top-1 accuracy at 50% of the quantized nodes, resulting in up to 96.88 MB of memory required to store the parameters. At this quantization level, one must consider whether the overhead of quantization and dequantization would not result in increased latency. FQ-ViT and static INT8 quantization introduced 0.92% and 3.07% quantization error for Swin-S/4/224, respectively. With our partial mixed-precision method, we reduced this quantization error to 0.32% with 90% of the nodes quantized.

Furthermore, as shown in Table 3.4, the quantization errors of ViT-B/8/224, ViT-B/16/224, and ViT-B/32/224 were reduced to $2.54\% \pm 1.27\%$ (80% nodes),

3. CHALLENGES IN POST-TRAINING QUANTIZATION OF VISION TRANSFORMERS

Table 3.5: Partial mixed-precision quantization results on ImageNet1K: This table presents the top-1 quantization error (mean $\pm$ standard deviation) for ViT and DeiT3 small and base models on the ImageNet1K validation set with input image size 384×384 in the top part. In the bottom part we present ViT, DeiT3, and Swin large and huge models. Results are averaged over ten runs. The lowest quantization error with the highest percentage of operations (Op.) quantized is highlighted in bold for each model.

Op. %	ViT-S/16/384	ViT-S/32/384	ViT-B/16/384	ViT-B/32/384	DeiT3-S/16/384	DeiT3-B/16/384
50%	1.91 $\pm$ 0.71	10.90 $\pm$ 6.16	15.53 $\pm$ 15.52	0.81 $\pm$ 0.26	2.76 $\pm$ 0.76	1.18 $\pm$ 0.30
55%	2.28 $\pm$ 0.39	7.99 $\pm$ 4.72	15.95 $\pm$ 15.93	1.08 $\pm$ 0.26	2.57 $\pm$ 0.88	1.17 $\pm$ 0.32
60%	2.43 $\pm$ 0.56	13.34 $\pm$ 4.50	21.31 $\pm$ 14.08	1.23 $\pm$ 0.34	2.79 $\pm$ 0.79	1.44 $\pm$ 0.39
65%	2.44 $\pm$ 0.44	12.36 $\pm$ 5.77	18.25 $\pm$ 14.74	1.28 $\pm$ 0.23	2.58 $\pm$ 0.62	1.46 $\pm$ 0.20
70%	2.73 $\pm$ 0.56	14.56 $\pm$ 5.33	28.65 $\pm$ 9.73	1.36 $\pm$ 0.34	2.79 $\pm$ 0.56	1.45 $\pm$ 0.28
75%	2.74 $\pm$ 0.78	11.72 $\pm$ 4.56	25.72 $\pm$ 12.92	1.37 $\pm$ 0.24	2.94 $\pm$ 0.68	1.60 $\pm$ 0.27
80%	3.08 $\pm$ 0.42	15.36 $\pm$ 4.58	22.51 $\pm$ 14.70	1.58 $\pm$ 0.16	2.99 $\pm$ 0.25	1.42 $\pm$ 0.30
85%	3.75 $\pm$ 0.25	19.32 $\pm$ 2.67	31.68 $\pm$ 0.92	1.86 $\pm$ 0.11	3.06 $\pm$ 0.65	1.58 $\pm$ 0.19
90%	3.65 $\pm$ 0.51	19.62 $\pm$ 5.26	29.07 $\pm$ 9.77	1.76 $\pm$ 0.18	3.18 $\pm$ 0.47	1.67 $\pm$ 0.15
95%	3.86 $\pm$ 0.46	20.66 $\pm$ 0.58	32.26 $\pm$ 1.22	1.99 $\pm$ 0.11	3.01 $\pm$ 0.14	2.42 $\pm$ 0.10
Op. %	ViT-L/16/224	ViT-L/16/384	DeiT3-L/16/224	DeiT3-L/16/384	DeiT3-H/14/224	Swin-L/4/224
50%	0.66 $\pm$ 0.20	0.73 $\pm$ 0.31	6.53 $\pm$ 17.56	0.72 $\pm$ 0.29	0.82 $\pm$ 0.35	0.34 $\pm$ 0.10
55%	0.69 $\pm$ 0.15	0.76 $\pm$ 0.23	0.81 $\pm$ 0.17	0.89 $\pm$ 0.47	0.82 $\pm$ 0.36	0.44 $\pm$ 0.05
60%	0.79 $\pm$ 0.14	0.93 $\pm$ 0.16	0.82 $\pm$ 0.20	0.89 $\pm$ 0.32	1.10 $\pm$ 0.24	0.42 $\pm$ 0.11
65%	0.86 $\pm$ 0.18	0.94 $\pm$ 0.13	1.01 $\pm$ 0.24	1.05 $\pm$ 0.32	1.69 $\pm$ 2.27	0.45 $\pm$ 0.02
70%	0.90 $\pm$ 0.19	1.00 $\pm$ 0.23	1.80 $\pm$ 2.38	1.20 $\pm$ 0.43	2.95 $\pm$ 5.31	0.45 $\pm$ 0.10
75%	0.94 $\pm$ 0.12	1.05 $\pm$ 0.24	1.64 $\pm$ 2.11	1.23 $\pm$ 0.36	3.22 $\pm$ 4.00	0.50 $\pm$ 0.06
80%	1.10 $\pm$ 0.17	1.08 $\pm$ 0.25	0.96 $\pm$ 0.19	1.38 $\pm$ 0.36	6.21 $\pm$ 5.27	0.49 $\pm$ 0.05
85%	1.14 $\pm$ 0.11	1.22 $\pm$ 0.18	1.10 $\pm$ 0.21	1.69 $\pm$ 0.23	5.04 $\pm$ 4.15	0.51 $\pm$ 0.06
90%	1.22 $\pm$ 0.17	1.25 $\pm$ 0.23	1.02 $\pm$ 0.08	1.70 $\pm$ 0.17	11.69 $\pm$ 7.54	0.51 $\pm$ 0.05
95%	1.34 $\pm$ 0.09	1.42 $\pm$ 0.11	9.63 $\pm$ 6.94	1.54 $\pm$ 0.30	14.51 $\pm$ 3.28	0.55 $\pm$ 0.06

8.90% $\pm$ 5.36% (80% nodes), and 4.21% $\pm$ 1.75% (50% nodes), respectively. The mixed precision quantization of ViT-B/8/224 achieved better top-1 accuracy than FQ-ViT. In addition, for DeiT-B/16/224 and DeiT-B/16/224/D, we reduced the quantization errors from 3.6% to 1.69% $\pm$ 0.06% (90% nodes) and 0.91% (FQ-ViT) to 0.42% $\pm$ 0.09% (90% nodes), respectively. For the Swin-B/4/224, our method reduced the quantization error from 10.93% (INT8) and 2.78% (FQ-ViT) to 1.01% $\pm$ 0.04% at 90% quantized nodes and 1.00% $\pm$ 0.10% at 95% nodes. At those quantization levels, the Swin-B/4/224 was compressed up to 32.5% and 28.75% compared to FP32 model’s weights, respectively. Interestingly, DeiT3-B/16/224 with INT8 static quantization and FQ-ViT lost all of its predictive power. However, our method reduced the quantization error to 1.8% at 80% nodes and 3.62% at 90% nodes.

Within the small and base models with larger input sizes (384x384), we observed a reduction in quantization error, except for ViT-B/16/384 and DeiT3-S/16/384, as

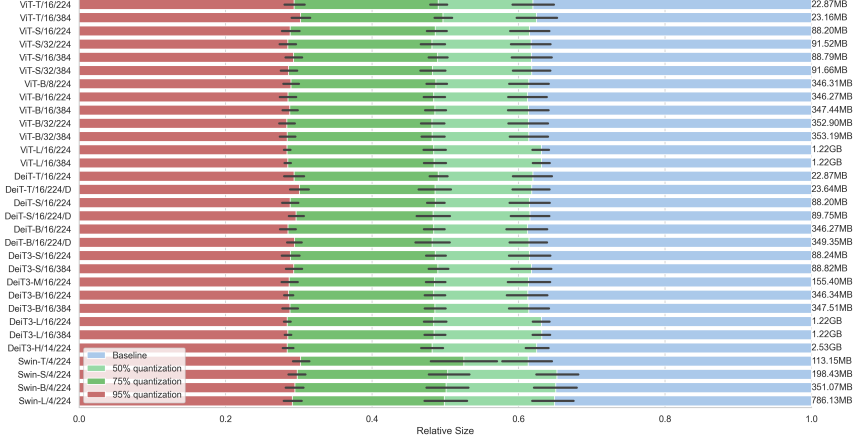


Figure 3.10: Effect of mixed-precision partial quantization on model memory size. Relative model size (x-axis) is shown for 50%, 75%, and 95% node quantization, with error bars indicating variation over ten runs. Baseline model sizes (FP32) are labeled on each bar.

shown in Table 3.5. ViT-S/32/384 reduced the quantization error from $20.73\% \pm 7.65\%$ to $14.56\% \pm 5.33\%$ for 70% quantized nodes. Interestingly, the quantization error for DeiT3-B/16/384 decreased significantly from $18.83\% \pm 23.35\%$ to $1.42\% \pm 0.30\%$ for 80% quantized nodes. In addition, we observed less variability affected by mixed precision quantization compared to DeiT3-B/16/224 using this model. In contrast, the ViT-B/16/384 quantization error increased from $8.90\% \pm 5.36\%$ to $22.51\% \pm 14.70\%$ at 80% nodes compared to ViT-B/16/224.

Finally, moving to the large and huge models in Table 3.5, we observed significant improvements in the top-1 accuracy of these models compared to INT8 static quantization, as shown in Table 3.1 and Table 3.2. Within the INT8 static quantization, Swin-L showed the most robust quantization with error equal to 4.99% compared to 72.96% and 84.63% for the ViT-L/16/224 and DeiT3-L/16/224, respectively.

With our method, we reduced the quantization error from 4.33% (INT8) and 0.71% (FQ-ViT) to $0.51\% \pm 0.05\%$ at 90% quantized nodes. In addition, the ViT-L models regained their predictive power even when 95% of the nodes were quantized. However, the best top-1 accuracy was achieved at 90% quantized nodes

with quantization errors of $1.22\% \pm 0.17\%$ and $1.25\% \pm 0.23\%$ for ViT-L/16/224 and ViT-L/16/384, respectively. Similarly, for the newer DeiT3-L models, we reduced the quantization error from 84.63% and 85.67% to $1.02\% \pm 0.08\%$ (90% nodes) and $1.38\% \pm 0.36\%$ at 80% quantized nodes. However, with DeiT3-L/16/224, we observed that at the 95% level, the quantization error increased to $9.63\% \pm 6.94\%$. Lastly, with DeiT3-H/14/224, we were able to reduce the model size by up to 90% of the nodes, which corresponds to an average of 32.5% of the required memory compared to the original for the FP32 model parameters, with the best mixed-precision quantization achieving 0.68% quantization error.

3.5 Summary and Conclusions

In this chapter, we focused on the challenges of post-training quantization of the ViT, DeiT, DeiT3, and Swin transformers. First, we showed that ViT and DeiT3 were more susceptible to quantization errors than DeiT and Swin with INT8 static quantization. Based on the model architecture and training differences, we hypothesized that the robustness was due to the regularization applied during training. In addition, we observed that the dynamic quantization did not affect the model’s performance. While FQ-ViT and PTQ4ViT are state-of-the-art methods, we showed that FQ-ViT failed to quantize base and large DeiT3 models. Finally, we observed that within the ViT family, overparameterization of the large models did not positively affect the robustness of the quantization.

Next, we analyzed the activation ranges of the fully connected layers in the MLP and attention blocks. We found that the outliers were present in all fully connected layers in all models. We also found, that for most of the fully connected layers, the outliers of the DeiT models were smaller in the distilled model than in its non-distilled counterpart. In addition, we observed that the ViT models usually had a more extensive outlier range than the DeiT distilled variants. Interestingly, DeiT-B/16/224 had one of the most significant outlier ranges in later blocks.

We observed similar findings within the layer’s sensitivity analysis using SQNR. Within all the models, we found that the first fully connected layer of the MLP block had a higher SQNR value for activations than the second fully connected layer. Moreover, the quantized weights to INT8 had a high SQNR value across all the layers and the models. This result might explain why the ViT, DeiT, DeiT3, and Swin Transformers are robust to dynamic quantization. Moreover, we showed a negative correlation between the average SQNR value of fully connected layers and the quantization error. We believe the SQNR metric can be valuable in assessing the model’s layers quantization robustness.

Finally, we proposed a simple yet effective mixed-precision post-training quantization method for the ViT, DeiT, DeiT3, and Swin Transformer models. We evaluated the models at different quantization thresholds, starting from 50% and going up to 95% of the quantized nodes. We showed that our method is competitive with FQ-ViT and PTQ4ViT while using only native quantized kernels. With our method, we were able to quantize the ViT, DeiT, DeiT3, and Swin Transformers up to 90% of the quantized nodes while keeping the model’s top-1 accuracy close to the FP baseline reference model. Furthermore, we believe that our method is portable across different hardware platforms. It could be used on mobile, embedded, CPU, or GPU platforms depending on the required model’s top-1 accuracy, latency, or size.

Chapter 4

Hybrid Quantization

4.1 Introduction

In Chapter 2 we defined quantization distinguishing between dynamic and static quantization, which we thoroughly evaluated in Chapter 3. Moreover, in Chapter 3, we presented the challenges of quantization of the ViT, DeiT, DeiT3, and Swin Transformer models. We also introduced a simple yet effective mixed-precision post-training quantization method that reduced quantization error while decreasing model size.

When choosing the quantization type for a model based on the target hardware, it is important to consider the device’s capabilities. In compute and memory constrained environments, static quantization is preferred, while dynamic quantization is the better option in memory-bound environments. For example, modern smartphones have powerful processors and dedicated tensor processing units, making them memory-bound [139, 140, 141] devices for many operations.

In this chapter, we introduce a novel algorithm - Hybrid Quantization (HQ) that integrates the strengths of both static and dynamic quantization for fully connected layers in ViT family models. By automatically combining these two approaches, our method aims to exploit the advantages of each: reducing latency compared to dynamic quantization while improving predictive power compared to static quantization. Specifically, we present a hybrid quantization algorithm that uses the SQNR metric to dynamically select between static and dynamic

4. HYBRID QUANTIZATION

quantization for linear layers, ensuring optimal performance. Our contributions are as follows:

- We introduce a novel HQ algorithm that combines static and dynamic quantization designed for a family of ViT models.
- We demonstrate that the predictive power of DeiT3 improves by 1%, 4%, and 11.5% for small, base, and large on the ImageNet1K validation dataset. Our method improves predictive power over statically quantized models in 12/12 ViT, 3/6 DeiT, 6/6 DeiT3, and 6/6 Swin Transformers.
- We measure the latency on diverse setups including different hardware:
 - iPhone 13 Pro with an A15 Bionic CPU: we report an average speedup latency of 1.23 and 1.28 for HQ1 and HQ3 on the A15 Bionic CPU compared to dynamic quantization for ViT models. Similarly, for DeiT3 family models, we achieve an average speedup of 1.33 and 1.29 for HQ1 and HQ3 on the same CPU.
 - CPU workstation with Intel Xeon 5218 Gold: we observe an average latency speedup of 1.13 for HQ1 and 1.15 for HQ3 in ViT models over dynamic quantization. For the DeiT3 models, we find an average speedup of 1.20 for HQ1 and 1.18 for HQ3 methods.
 - Lastly, we evaluate on NVIDIA A100 GPU-equipped workstation. Compared to dynamic quantization, we achieve an average speedup of 1.68 for ViT models, 1.72 for DeiT3 models for HQ1, and 1.69 for HQ3.

4.2 Related Work

Most of the literature on the post-training quantization of DL models focuses on one aspect of the quantization: static, dynamic, or mixed precision. Nevertheless, the term hybrid quantization was independently adopted in the literature to describe various quantization methods. Nazari and Salehi’s hybrid quantization mixed fixed-point and power-of-two static quantization [142]. Their method was applied to ResNet-18 [35], and a quantized model was deployed on a field-programmable gate array (FPGA). Panda proposed a method that assigned different bit widths to convolutional layers based on the defined adversarial noise sensitivity [143]. The method was described as hybrid quantization. Nevertheless, in modern literature, it would be called mixed-precision quantization. Pandey et al. utilized the SQNR metric to apply mixed-precision static quantization to DL models [144]. They

explore the bit widths ranging from 4 bits up to 16 bits for weights and activations. Moreover, they showed that their method integrates with AdaRound.

While there is ongoing research on ViT quantization, most studies focus on static quantization with mixed precision. In addition, they introduce approximations to the activation functions and layer normalization, that require custom kernel implementations for each platform. To the best of our knowledge, we are the first to propose a hybrid approach that combines static and dynamic quantization in ViT models.

4.3 Hybrid Quantization

The ViT models are a specific type of feedforward deep neural network. As explained in Section 2.1, these models can be represented as directed acyclic graphs that sequentially process their input X through various nodes (or layers) denoted as $f \in f^1, f^2, \dots, f^m$ and ultimately produce a tensor Y . Each node represents a distinct operation in the ViT model, such as matrix multiplication, LayerNorm [18], GELU activation [19], and others. Our objective is to statically quantize specific nodes, such as normalization layers, activation functions, and attention modules. However, linear layers can be quantized statically (INT8) or dynamically (INT8-D). Like in dynamic quantization, skip connections are maintained in floating-point arithmetic to facilitate the signal flow through the network.

Tuning phase

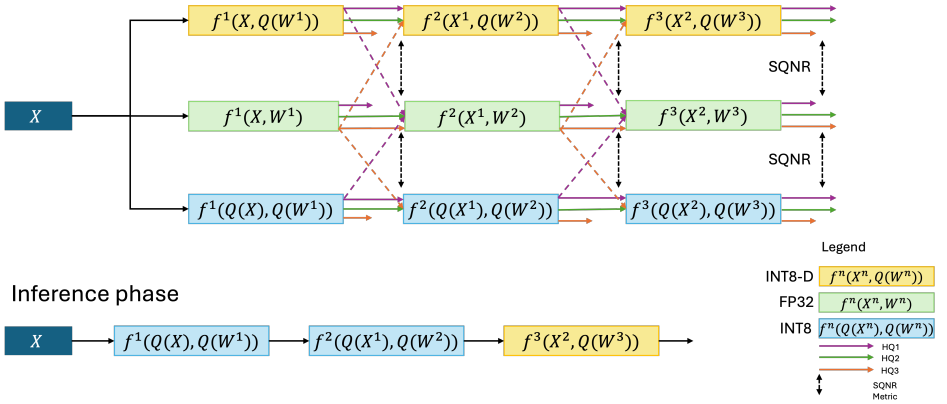


Figure 4.1: A high-level overview of the HQ algorithms: during the tuning phase, the signal-to-noise ratio (SQNR) is calculated between the outputs of the models at the same linear node (f). The resulting model is a combination of static and dynamic quantized blocks used in the inference phase.

4. HYBRID QUANTIZATION

We have developed three different hybrid quantization algorithm variations based on SQNR measurements Equation (2.13):

- Hybrid Quantization 1 (HQ1) measures the impact of the quantized output signal compared to the linear layer’s baseline signal. We need to dequantize the intermediate activations before passing them to the reference model. This allows us to measure the quantization’s impact on the specific linear layer.
- HQ2 involves passing both the quantized and original signals through their respective networks independently, i.e., the signal flows uninterrupted through a single neural network. The signal measured here tells us how the quantization affects the signal up to the given linear layer without any quantization side effect.
- HQ3 quantizes the signal from a baseline deep neural network preceding linear layer operation and then passes it through a quantized neural network linear layer. As a result, we can measure how INT8 or INT8-D quantization at the given linear layer affects the signal strength from the baseline FP32 model.

Figure 4.1 provides a high-level overview of these methods, illustrating the flow of signals through the networks and a simplified model overview used for inference. Each SQNR measurements are done at the output of the linear layers. Specific implementation details are given in Algorithm 1.

To reiterate, HQ1 allows us to assess the influence of the quantized output on the baseline model’s signal. At the same time, HQ3 involves passing the signal from the baseline model to the quantized model and measuring the effect of quantization on the signal in the quantized model. Finally, HQ2 allows us to measure how independent signal processing by the networks affects the signal between the quantized and baseline models.

4.4 Experimental Setup

In this section, we describe the experimental setup: DL models, dataset, metrics, and hardware. We utilized, as previously, the ViT, DeiT, Swin, and DeiT3 models on the ImageNet1K dataset [22]. We report the top-1 accuracy on the validation set while utilizing the training dataset for calibration and measuring the SQNR metric. The pre-trained FP models were fetched from the *timm* package [138] and converted to static and dynamic quantized models. In our setup, we considered scalar symmetric quantization, i.e., singular scale value per weight and activation

matrix. We applied the following observers and data types to compute the scaling factor:

- weights: absolute min-max quantization¹; INT8,
- activations: running histogram observer²; UINT8.

Firstly, we converted each model to a baseline static and dynamic quantized version as a starting point for HQ algorithms. Within static quantized baseline models we kept the skip connection in FP arithmetic. We evaluated our HQ methods five times, starting from the same baseline. We randomly sampled 2000 samples for each run to measure SQNR and calibration from the ImageNet1K training set. We reported the best result and the mean and standard deviation over five runs. Our results were compared with the FQ-ViT method [96].

We additionally measured the latency of the HQ methods on three various hardware environments:

- iPhone 13 Pro with Apple A15 Bionic CPU,
- Baremetal CPU with Intel Xeon 5218 Gold, and
- GPU-equipped workstation with an NVIDIA A100 80GB GPU.

We utilized quantized kernels implemented in FBGEMM [145] and QNNPACK [146] in PyTorch [122] and PyTorch Mobile to compute latency on Intel Xeon 5218 Gold and Apple A15 Bionic CPU. For the GPU-equipped workstation, we implemented GPU-quantized kernels using Triton [131]. Latency measurements were obtained by running 1000 random samples with input image sizes of 224x224 or 384x384, depending on the model size. The batch size was set to one to reflect an online (streaming) inference scenario. We report the latency plots per model per HQ algorithm compared to static and dynamic quantization. We also report the average speedup over five runs compared to dynamic quantization.

Finally, we plot the accuracy vs. latency tradeoff plots for selected models that together represent the full range of the ViT, DeiT, and DeiT3 architectures. We plot the mean and standard deviation for each HQ algorithm, as well as the best model out of five runs. We compared this with the static and dynamic quantization accuracy and latency. We marked the boundaries - top-1 accuracy and latency - of static and dynamic quantization with a dashed line.

¹<https://github.com/pytorch/pytorch/blob/be96ccf77c460efc85d281efd501601368d90b55/torch/ao/quantization/observer.py#L413>

²<https://github.com/pytorch/pytorch/blob/be96ccf77c460efc85d281efd501601368d90b55/torch/ao/quantization/observer.py#L956>

4.5 Results

Figure 4.2 presents the mean and standard deviation of dynamic to static quantized linear layers ratio depending on the model and the HQ algorithm. We found that, when our HQ algorithms were applied to the Swin Transformer, it chose static over dynamic quantization for linear layers. This aligns with our findings in Chapter 3, as the Swin Transformer models were robust to the static quantization. We improved the top-1 accuracy over the static quantization with our HQ algorithm on the ImageNet1K validation dataset. Nevertheless, we recommend that Swin Transformers remain statically quantized with optional skip connections in FP arithmetic. As a result, we did not compute latency speedups for this model’s architecture compared to the dynamic quantization.

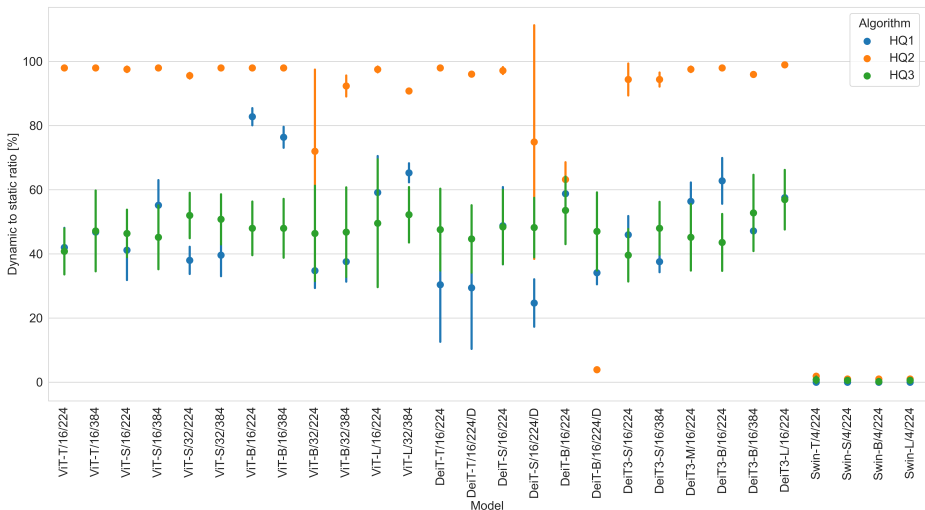


Figure 4.2: Percentage of dynamic quantized layers in the models: The X-axis displays the model, while the Y-axis represents the ratio of dynamic to static layers. Higher values indicate a higher percentage of dynamic layers in the model after HQ. The points on the graph represent the average ratio over five runs, with the standard deviation plotted.

We also found that the HQ2 algorithm chose dynamic quantization for linear layers for the majority of models, except for DeiT-B/16/224 and DeiT-B/16/224/D. The HQ2 algorithm, on average, selected $94.50\% \pm 9.87\%$ for the ViT family, $75.35\% \pm 34.53\%$ for DeiT, and $96.56\% \pm 2.72\%$. While with the HQ1 algorithm, we

observed a more conservative approach to selecting statically quantized linear, i.e., $51.57\% \pm 16.67\%$ for ViT, $37.87\% \pm 15.34\%$ for DeiT, and $51.26\% \pm 10.15\%$ for DeiT3 models.

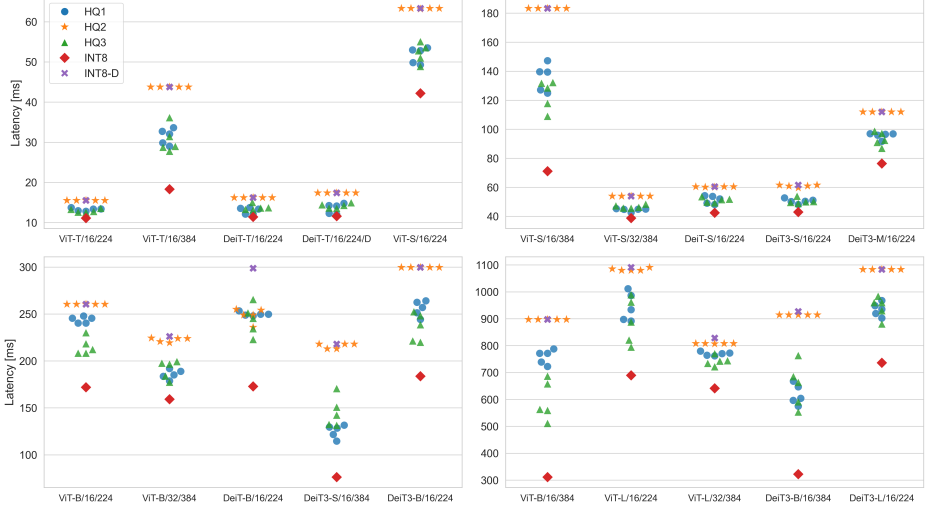


Figure 4.3: Average model inference latency on the A15 Bionic CPU: each point represents a single quantization configuration of the selected HQ algorithm. We compared the latency of the proposed methods with INT8 and INT8-D. The measurements were averaged over 1000 samples.

Our proposed HQ methods overall improved the predictive power over INT8 static quantization in 12/12 ViT, 3/6 DeiT, 6/6 DeiT3, and 6/6 Swin Transformers, as observed in Table 4.1 and Table 4.2 on the ImageNet1K validation dataset. While FQ-ViT achieved better overall performance in ViT models, unfortunately, it did not transfer to a newer architecture such as DeiT3, especially for base and large models. Likewise, we decreased inference latency when our method was applied relative to dynamic quantization on 12/12 ViT, 6/6 DeiT, and 6/6 DeiT3 models as presented in the Table 4.3.

Within the ViT family, we observed that, on average, the HQ3 method improved the top-1 accuracy of the models more than the HQ1 method on the ImageNet1K validation dataset. However, for ViT-S/32/224 and ViT-S/32/384, the highest increase in top-1 accuracy was recorded with the HQ1 algorithm. In addition, as shown in Table 4.3, the latency improvements over dynamic quantization were observed for the same models on the A15 mobile CPU and an NVIDIA A100 GPU with the HQ1 method. However, on average, we observed 1.28, 1.15, and 1.68 times

4. HYBRID QUANTIZATION

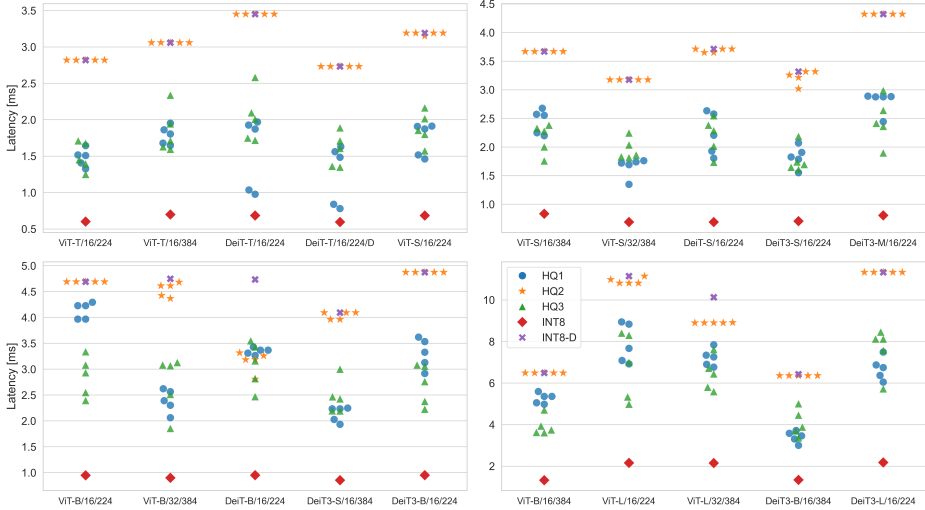


Figure 4.4: Average inference latency of models on the A100 GPU-equipped workstation: each point represents a single quantization configuration of the selected HQ algorithm. We compared the latency of the proposed methods with INT8 and INT8-D. The measurements were averaged over 1000 samples.

lower latency with HQ3 compared to dynamic quantization for the A15 mobile CPU, Intel Xeon 5218 Gold CPU, and NVIDIA A100 GPU, respectively.

As for the HQ2 within the ViT family, we observed increased quantization errors compared to the dynamic quantization with no decrease in latency. The only model that moderately benefited from HQ2 was ViT-B/32/224, which improved top-1 accuracy by 3.64% on the ImageNet1K validation dataset over static quantization and reduced latency by up to 1.13 and 1.26 on average on the mobile A15 CPU and NVIDIA A100 GPU, respectively.

In addition, in Figures 4.3 to 4.5, we present the average inference time for all runs and methods across three hardware setups: mobile A15 CPU, NVIDIA A100 GPU, and Intel Xeon 5218 Gold CPU, respectively. Within the ViT family, as demonstrated by the static to dynamic linear ratio, we found that the latencies of the HQ1 and HQ3 methods for inference with a batch size equal to one of the model’s latencies are reduced compared to dynamic (INT8-D) quantization while being slower than static quantization (INT-8). As expected, our HQ2 method, which selects almost every layer for dynamic quantization, approaches the latency of dynamic quantization.

Between the three hardware variations, we found similar patterns in the latency

across the ViT family. We observed that for ViT-S/16/384, HQ3 achieved better latency than HQ1. Similarly, for ViT-B/16/224 and ViT-B/16/384, the faster inference was obtained by utilizing the HQ3 algorithm. However, for ViT-B/32/224, lower latency was achieved by the HQ1 algorithm. Lastly, for the ViT-L/16/224, the lowest latency was obtained by applying the HQ3 algorithm on all platforms.

In Figures 4.6 and 4.7, we show the tradeoff between accuracy and latency of our methods for ViT-S/32/224, ViT-B/32/384, and ViT-L/32/384. We have marked the quantization spectra with red dots - static (INT8) and dynamic (INT8-D) quantization. The horizontal dashed red line marks the boundary of the top-1 accuracy of static quantization, while the vertical line marks the latency of the dynamic quantization. With HQ1 and HQ3 on average we improved the top-1 accuracy and latency over static and dynamic quantization for ViT-S/32/224. Moreover, the best models (marked with blue and green diamonds for HQ1 and HQ3, respectively) outperformed static and dynamic quantization in top-1 accuracy and latency, respectively. Meanwhile, the HQ2 algorithm achieved the same latency on average as dynamic quantization with worse top-1 accuracy.

Only the best HQ1 model achieved a satisfactory tradeoff between accuracy and latency for ViT-B/32/384, while it underperformed on average in the top-1 accuracy on the ImageNet1K validation dataset. Nevertheless, we observed



Figure 4.5: CPU-only workstation: Average Intel Xeon 5218 Gold CPU inference latency of models. Each point represents a single quantization configuration. We report the average latency of linear layers over 1000 samples.

4. HYBRID QUANTIZATION

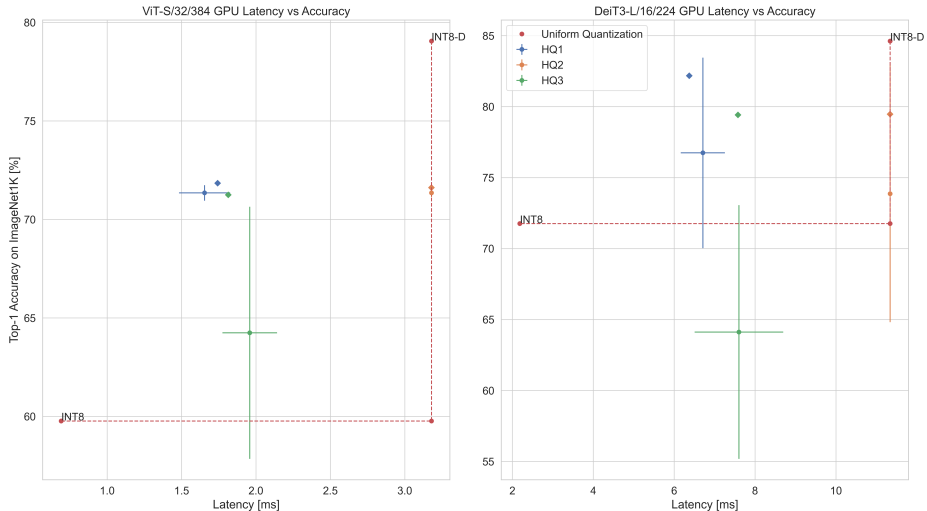


Figure 4.6: Latency vs. Accuracy Tradeoff of HQ Algorithms ViT-S/32/384 and DeiT3-L/16/224: This figure shows the tradeoff between latency and accuracy of the HQ algorithms compared to static (INT8) and dynamic (INT8-D) quantization. For each HQ variation, we plot the mean and standard deviation with a diamond, and we plot the best model out of five runs. Measurements were taken on an NVIDIA A100 GPU. The results above and to the left of the dashed red line indicate the improvement in accuracy over static quantization and latency over dynamic quantization, respectively.

improved speedups over dynamic quantization. Interestingly, the HQ2 method lost most predictive power for this model while marginally improving latency compared to static and dynamic quantization. Similarly, for ViT-L/32/384, the best results were obtained by the HQ1 and HQ3 variants, which improved the latency and top-1 accuracy compared to static and dynamic quantization. However, on average, static quantization achieved better top-1 accuracy than HQ3 on the ImageNet1K validation dataset.

The DeiT family of models as well as the Swin transformers previously showed robustness to quantization. In our experiments, we improved performance over static quantization and reduced latency over dynamic quantization. In Figures 4.3 to 4.5, we observed that the inference latency of HQ1 for DeiT-T models approached that of static quantization. However, the static quantization with FP skip connection achieved better top-1 accuracy and faster inference time than the HQ1 method. Similarly, the HQ2 method selected dynamic quantization over static

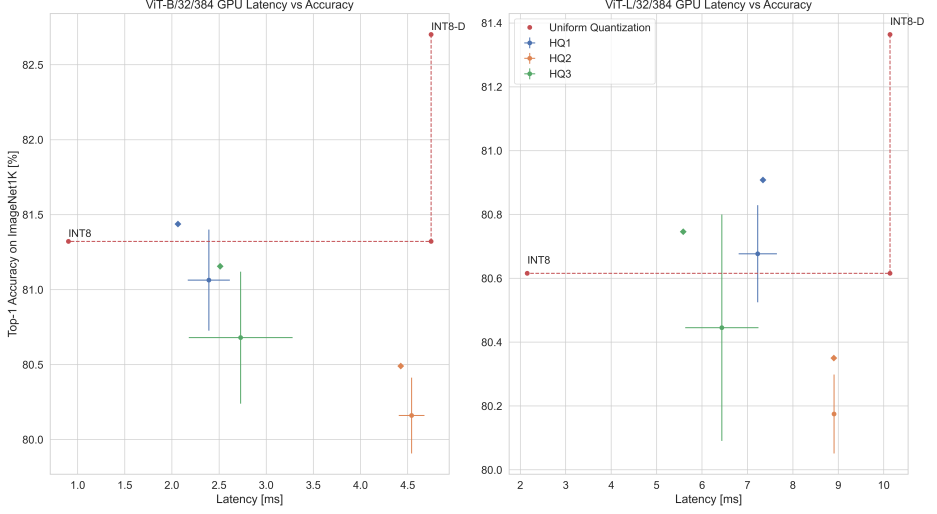


Figure 4.7: Latency vs. Accuracy Tradeoff of HQ Algorithms ViT-B/32/384 and ViT-L/32/384: This figure shows the tradeoff between latency and accuracy of the HQ algorithms compared to static (INT8) and dynamic (INT8-D) quantization. For each HQ variation, we plot the mean and standard deviation with a diamond, and we plot the best model out of five runs. Measurements were taken on an NVIDIA A100 GPU. The results above and to the left of the dashed red line indicate the improvement in accuracy over static quantization and latency over dynamic quantization, respectively.

quantization for the linear layers in the DeiT-T model. As a result, the latency approached that of the dynamic quantization, and the top-1 accuracy decreased compared to both static and dynamic quantization on the ImageNet1K validation dataset.

For the DeiT-S/16/224 model, the average latency was reduced by up to 1.14 and 1.15 for HQ1 and HQ3, respectively, compared to dynamic quantization. Meanwhile, for DeiT-S/16/224/D, we observed an even more significant latency reduction of 1.23 for HQ1 and 1.14 for the HQ3 algorithm. For DeiT-S/16/224, the best improvement in the latency and top-1 accuracy was obtained by applying the HQ3 method. Nevertheless, we recommend utilizing static quantization with FP skip connections for this model as it outperformed the HQ1 and HQ3 methods by a small margin.

Finally, as mentioned above, DeiT-B/16/224 and DeiT-B/16/224/D were marked as outliers in the HQ2 static to dynamic quantization layer ratio. The

4. HYBRID QUANTIZATION

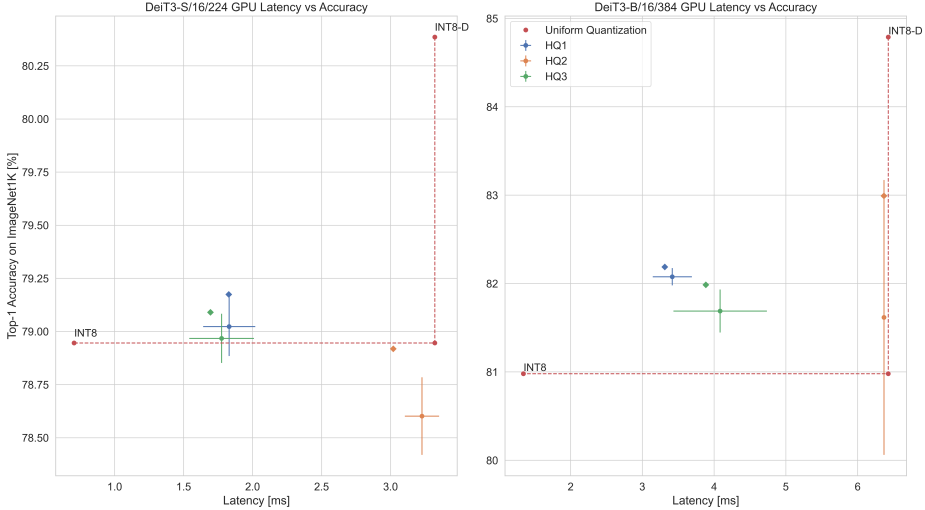


Figure 4.8: Latency vs. Accuracy Tradeoff of HQ Algorithms DeiT-S/16/224 and DeiT3-B/16/384: This figure shows the tradeoff between latency and accuracy of the HQ algorithms compared to static (INT8) and dynamic (INT8-D) quantization. For each HQ variation, we plot the mean and standard deviation with a diamond, and we plot the best model out of five runs. Measurements were taken on an NVIDIA A100 GPU. The results above and to the left of the dashed red line indicate the improvement in accuracy over static quantization and latency over dynamic quantization, respectively.

former achieved average speedups of up to 1.11, 1.14, and 1.14 over dynamic quantization, while the latter achieved 1.23, 1.36, and 1.15 for HQ1, HQ2, and HQ3, respectively. DeiT-B/16/224 achieved the best top-1 accuracy with the FQ-ViT method compared to static quantization. Nevertheless, the best models obtained by our HQ1 and HQ3 methods also improved the top-1 accuracy on the ImageNet1K validation dataset. Interestingly, the best top-1 accuracy was obtained with the HQ2 method for DeiT-B/16/224/D.

With the HQ1 algorithm, we noted an average of $79.30\% \pm 0.12\%$ and $80.78\% \pm 0.07\%$ top-1 accuracy for DeiT3-S/16/224 and DeiT3-S/16/384 on the ImageNet1K validation dataset; see Table 4.1. The best HQ results were achieved at 79.42% and 80.89% with HQ1 compared to static quantization at 78.95% and 80.39%, respectively. The best top-1 accuracy was achieved with FQ-ViT: 79.62% and 81.28% for DeiT3-S/16/224 and DeiT3-S/16/384; see Table 4.2. The highest average speedups for these models were observed with the HQ1 and HQ3 methods on the

NVIDIA A100 GPU. For DeiT3-S/16/224, we observed 1.81 and 1.87 times lower latency compared to dynamic quantization. Meanwhile, for the DeiT3-S/16/384, we observed speedups of 1.91 and 1.67 compared to dynamic quantization; see Table 4.3 and Figure 4.8. Similarly to ViT and DeiT models, we did not observe significant speedups with HQ2 compared to dynamic quantization.

In the case of DeiT3-M/16/224, the most significant improvement over static quantization was achieved using FQ-ViT, followed by HQ1. Surprisingly, the next best top-1 accuracy was achieved by using the HQ2 method. Overall, however, we did not observe any speedup over the three hardware variants compared to dynamic quantization. The HQ3 method yielded the most significant average speedup: 1.20 on the mobile device, 1.15 on the Intel Xeon 5218 Gold CPU, and 1.76 on the NVIDIA GPU. Using HQ1, we achieved 1.17, 1.14, and 1.55 speedups on the same hardware devices; see Table 4.3.

Starting with the DeiT3-B/16/224, we observed that FQ-ViT strongly affected the top-1 accuracy of the models. At the same time, our methods maintained the predictive power of these models. For DeiT3-B/16/224, we obtained a top-1 accuracy of 80.81% top-1 compared to 76.54% for static quantization; see Table 4.2. However, we found high top-1 accuracy variability with HQ1, averaging $61.78\% \pm 25.13\%$. Lower variability was observed for the HQ3 with $70.58\% \pm 18.00\%$. However, the best model achieved 80.15% top-1 accuracy on the ImageNet1K validation dataset. Interestingly, the HQ2 best model achieved a better accuracy of 80.40% than the 80.15% for HQ3. In addition, for a mobile A15 CPU, an Intel Xeon 5218 Gold CPU, and an NVIDIA A100 GPU, we obtained an average speedup over dynamic quantization of 1.27, 1.19, and 1.80, respectively. Meanwhile, no average speedup was observed for the HQ2 method.

Surprisingly, the best top-1 accuracy was obtained by the HQ2 method for DeiT3-B/16/384 with 82.99% top-1 accuracy. Moreover, compared to the DeiT3-B/16/224 across all methods, we observed low top-1 accuracy variability, i.e., $82.08\% \pm 0.09\%$, $81.62\% \pm 1.39\%$, and $81.69\% \pm 0.22\%$ for HQ1, HQ2, and HQ3, respectively. As shown in Table 4.3 and Figure 4.8, the best latency improvements were observed with the HQ1 algorithm.

Lastly, our HQ1 method improved the DeiT3-L/16/224 top-1 accuracy by 11.5% compared to static quantization. At the same time, we observed an average speedup of 1.16, 1.16, and 1.69 for the mobile A15 CPU, an Intel Xeon 5218 Gold CPU, and an NVIDIA A100 GPU, respectively. However, we observed variability over five runs of $72.26\% \pm 7.16\%$, $71.48\% \pm 7.52\%$, and $73.07\% \pm 4.55\%$ for the HQ1, HQ2, and HQ3 methods. In addition, as shown in Figure 4.6, the HQ1 method, on average, outperformed the top-1 accuracy of static quantization while reducing latency compared to dynamic quantization.

4. HYBRID QUANTIZATION

Table 4.1: Average top-1 accuracy with standard deviation on the ImageNet1K validation dataset computed over five runs. Our method improved the top-1 accuracy over the reference INT8 static model in 12/12 ViT, 3/6 DeiT, 6/6 DeiT3, and 4/4 Swin models.

Model	HQ1	HQ2	HQ3
ViT-T/16/224	8.32±0.62	12.25±0.13	10.84±1.51
ViT-T/16/384	7.75±1.52	10.01±0.82	8.22±1.75
ViT-S/16/224	76.80±0.21	73.12±0.21	74.70±1.20
ViT-S/16/384	76.60±0.20	76.99±0.08	78.63±1.31
ViT-S/32/224	60.26±0.47	59.24±0.15	54.27±7.19
ViT-S/32/384	71.35±0.35	71.35±0.19	64.25±5.73
ViT-B/16/224	68.64±2.38	69.81±0.34	71.59±1.87
ViT-B/16/384	43.83±4.17	46.17±1.90	48.00±2.23
ViT-B/32/224	73.81±1.35	74.00±1.07	72.65±1.33
ViT-B/32/384	81.06±0.30	80.16±0.23	80.68±0.39
ViT-L/16/224	83.80±0.09	84.01±0.14	84.19±0.24
ViT-L/32/384	80.68±0.14	80.18±0.11	80.45±0.32
DeiT-T/16/224	71.40±0.08	70.69±0.05	71.13±0.22
DeiT-T/16/224/D	73.88±0.12	73.30±0.02	73.71±0.16
DeiT-S/16/224	78.41±0.15	78.18±0.14	78.53±0.15
DeiT-S/16/224/D	80.78±0.07	80.21±0.23	80.48±0.14
DeiT-B/16/224	78.28±0.09	78.07±0.08	78.48±0.21
DeiT-B/16/224/D	82.83±0.02	82.89±0.04	81.77±0.85
DeiT3-S/16/224	79.30 ± 0.12	78.77 ± 0.10	79.09 ± 0.12
DeiT3-S/16/384	80.78 ± 0.07	80.69 ± 0.07	80.68 ± 0.08
DeiT3-M/16/224	79.60 ± 0.10	79.55 ± 0.03	79.42 ± 0.12
DeiT3-B/16/224	61.78 ± 25.13	58.83 ± 24.69	70.58 ± 18.00
DeiT3-B/16/384	82.08 ± 0.09	81.62±1.39	81.69±0.22
DeiT3-L/16/224	72.26 ± 7.16	71.48 ± 7.52	73.07 ± 4.55
Swin-T/4/224	79.27±0.04	79.24±0.08	79.17±0.11
Swin-S/4/224	82.65±0.03	82.67±0.04	82.64±0.03
Swin-B/4/224	84.10±0.03	84.15±0.04	84.13±0.05
Swin-L/4/224	85.66±0.07	85.71±0.04	85.67±0.05

Table 4.2: Comparison between baseline static quantization (INT8), dynamic quantization (INT8-D), hybrid quantization (HQ1, HQ2, and HQ3), and FQ-ViT. We report top-1 accuracy on the ImageNet1K validation dataset. In this experiment we present the best model out of 5 runs. We reproduced and extended the results of FQ-ViT. In bold we mark the best result between HQ and FQ-ViT.

Model	INT8	INT8-D	HQ1	HQ2	HQ3	FQ-ViT
ViT-T/16/224	9.02	70.94	9.27	12.41	13.18	45.80
ViT-T/16/384	7.65	74.45	9.90	11.05	10.55	45.18
ViT-S/16/224	77.01	79.27	77.12	73.43	77.05	78.58
ViT-S/16/384	79.40	82.36	76.81	77.08	79.77	81.61
ViT-S/32/224	45.59	73.95	60.93	59.43	60.58	59.13
ViT-S/32/384	59.77	78.70	71.84	71.62	71.25	68.43
ViT-B/16/224	73.15	83.96	71.76	70.27	74.89	83.70
ViT-B/16/384	51.05	85.03	48.82	48.75	51.78	84.11
ViT-B/32/224	71.35	79.13	75.10	74.99	74.57	70.99
ViT-B/32/384	81.32	82.64	81.44	80.49	81.16	81.37
ViT-L/16/224	84.30	85.29	83.94	84.18	84.40	84.97
ViT-L/32/384	80.62	81.36	80.91	80.35	80.75	81.36
DeiT-T/16/224	71.63	71.56	71.51	70.76	71.43	71.05
DeiT-T/16/224/D	74.16	73.99	74.03	73.31	73.94	73.63
DeiT-S/16/224	78.84	78.91	78.70	78.32	78.74	78.49
DeiT-S/16/224/D	80.75	80.64	80.86	80.66	80.68	80.42
DeiT-B/16/224	78.23	81.29	78.40	78.17	78.67	80.96
DeiT-B/16/224/D	82.92	82.44	82.87	82.96	82.51	82.48
DeiT3-S/16/224	78.95	80.63	79.42	78.93	79.26	79.62
DeiT3-S/16/384	80.39	82.91	80.89	80.81	80.80	81.28
DeiT3-M/16/224	79.47	82.74	79.72	79.59	79.56	82.1
DeiT3-B/16/224	76.54	83.51	80.81	80.40	80.15	0.10
DeiT3-B/16/384	80.98	84.79	82.19	82.99	81.99	0.10
DeiT3-L/16/224	71.77	84.61	83.27	82.01	80.29	0.10
Swin-T/4/224	79.19	80.84	79.33	79.36	79.28	80.02
Swin-S/4/224	80.70	83.20	82.69	82.74	82.69	82.40
Swin-B/4/224	84.19	84.99	84.16	84.21	84.18	82.50
Swin-L/4/224	85.72	86.15	85.77	85.78	85.75	85.61

4. HYBRID QUANTIZATION

Table 4.3: Average speedup latency improvement: this table presents the average speedup latency computed over five runs of the model compared to dynamic quantization. Each HQ configuration was executed on three hardware devices: iPhone 13 Pro with A15 CPU, baremetal with Intel Xeon 5218 Gold CPU, and a server with an NVIDIA A100 80GB GPU. The best speedup is shown in bold.

Model	A15 CPU			Intel Xeon CPU			A100 GPU		
	HQ1	HQ2	HQ3	HQ1	HQ2	HQ3	HQ1	HQ2	HQ3
ViT-T/16/224	1.17	1.00	1.20	1.19	1.00	1.20	1.90	1.00	1.88
ViT-T/16/384	1.39	1.00	1.43	1.17	1.00	1.19	1.71	1.00	1.66
ViT-S/16/224	1.23	1.00	1.21	1.19	1.00	1.14	1.84	1.00	1.70
ViT-S/16/384	1.35	1.00	1.48	1.18	1.00	1.23	1.49	1.00	1.71
ViT-S/32/224	1.41	1.00	1.28	1.04	1.01	1.03	2.14	1.03	1.63
ViT-S/32/384	1.21	1.00	1.16	1.12	1.00	1.14	1.92	1.00	1.62
ViT-B/16/224	1.07	1.00	1.21	1.03	1.00	1.10	1.13	1.00	1.64
ViT-B/16/384	1.18	1.00	1.51	1.09	1.05	1.21	1.23	1.00	1.65
ViT-B/32/224	1.36	1.13	1.32	1.01	1.01	1.09	1.97	1.26	1.72
ViT-B/32/384	1.22	1.02	1.18	1.18	1.01	1.15	1.99	1.05	1.74
ViT-L/16/224	1.16	1.01	1.22	1.11	1.01	1.15	1.41	1.02	1.64
ViT-L/32/384	1.08	1.03	1.12	1.09	1.04	1.12	1.40	1.14	1.64
ViT - Avg. all	1.23	1.01	1.28	1.13	1.01	1.15	1.68	1.04	1.68
DeiT-T/16/224	1.25	1.00	1.18	1.24	1.00	1.15	2.21	1.01	1.70
DeiT-T/16/224/D	1.25	1.00	1.23	1.26	1.00	1.19	2.16	1.01	1.73
DeiT-S/16/224	1.17	1.00	1.18	1.14	1.00	1.15	1.66	1.00	1.66
DeiT-S/16/224/D	1.43	1.61	1.25	1.23	1.06	1.14	2.51	1.22	1.66
DeiT-B/16/224	1.20	1.20	1.23	1.11	1.14	1.14	1.41	1.50	1.53
DeiT-B/16/224/D	1.44	1.01	1.24	1.23	1.36	1.15	2.15	4.3	1.65
DeiT - Avg. all	1.29	1.15	1.21	1.20	1.09	1.15	2.02	1.67	1.66
DeiT3-S/16/224	1.22	1.01	1.22	1.18	1.01	1.18	1.81	1.00	1.87
DeiT3-S/16/384	1.74	1.01	1.50	1.31	1.00	1.23	1.91	1.01	1.67
DeiT3-M/16/224	1.17	1.00	1.20	1.14	1.00	1.15	1.55	1.00	1.76
DeiT3-B/16/224	1.17	1.00	1.27	1.14	1.00	1.19	1.47	1.00	1.80
DeiT3-B/16/384	1.50	1.01	1.42	1.27	1.01	1.20	1.88	1.00	1.57
DeiT3-L/16/224	1.16	1.00	1.15	1.16	1.00	1.13	1.69	1.00	1.49
DeiT3 - Avg. all	1.33	1.01	1.29	1.19	1.00	1.18	1.72	1.01	1.69

Algorithm 4.1 Post-training Hybrid Quantization algorithms. The specific operations in the HQ1, HQ2, and HQ3 algorithms are shown in the tables below. The remaining steps are common to all variants.

Require:

The dataset $D_s = \{s_1, s_2, \dots, s_i\}$ containing i samples
 The calibration dataset $D_c = \{s_1, s_2, \dots, s_j\}$ containing j calibration samples
 Neural network ViT = $\{f^1, f^2, \dots, f^m\}$ consisting of m nodes
 Static quantization function $\mathcal{Q}_S$; Dynamic quantization function $\mathcal{Q}_D$
 Dequantize function $\hat{\mathcal{Q}}$; Calibration function calibrate

ViT_S $\leftarrow \mathcal{Q}_S(\text{calibrate}(\text{ViT}, D_c))$; ViT_D $\leftarrow \mathcal{Q}_D(\text{ViT})$

Initialize SQNR list L

for $s_i \in D_s$ **do**

$Y_S \leftarrow \mathcal{Q}_S(s_i)$; $Y_D \leftarrow \mathcal{Q}_D(s_i)$; $Y \leftarrow s_i$

for $f^k \in \text{ViT}$, $f_S^k \in \text{ViT}_S$, $f_D^k \in \text{ViT}_D$ **do**

HQ1	HQ2	HQ3
$Z_S \leftarrow f^k(\hat{\mathcal{Q}}(Y_S))$	$Y_S \leftarrow f_S^k(Y_S)$	$Y_S \leftarrow f_S^k(\mathcal{Q}_S(Y))$
$Y_S \leftarrow f_S^k(Y_S)$	$Y_D \leftarrow f_D^k(Y_D)$	$Y_D \leftarrow f_D^k(\mathcal{Q}_D(Y))$
$Z_D \leftarrow f^k(\hat{\mathcal{Q}}(Y_D))$	$Y \leftarrow f^k(Y)$	
$Y_D \leftarrow f_D^k(Y_D)$		

if f^k is a linear layer **then**

HQ1	HQ2 & HQ3
$\omega^S \leftarrow \text{SQNR}_S(Z_S, Y_S)$	$\omega^S \leftarrow \text{SQNR}_S(Y, Y_S)$
$\omega^D \leftarrow \text{SQNR}_D(Z_D, Y_D)$	$\omega^D \leftarrow \text{SQNR}_D(Y, Y_D)$

Store $(f^k, \omega^S, \omega^D)$ in L

end if

end for

end for

$L_G \leftarrow$ Group the list L by f^k

for $f^k, \omega^S, \omega^D \in L_G$ **do**

$\bar{\omega}^S \leftarrow \frac{1}{n} \sum_{i=1}^n \omega_i^S$; $\bar{\omega}^D \leftarrow \frac{1}{n} \sum_{i=1}^n \omega_i^D$

if $\bar{\omega}^S \geq \bar{\omega}^D$ **then**

Select f^k for static quantization

else

Select f^k for dynamic quantization

end if

end for

4.6 Summary and Conclusions

In this chapter, we introduced a novel post-training hybrid quantization method: HQ. Due to the properties of the ViT architecture, we could introduce and mix static and dynamic quantization to the linear layers while keeping the rest of the operations, such as activations, normalizations, and attention, statically quantized. Our algorithm automatically selected the quantization scheme, between static or dynamic, based on the SQNR metric between static and dynamic quantization. Furthermore, our method is efficient and can be tuned on a single A100 GPU. Depending on the size of the model, it takes minutes for tiny and small models to a couple of hours for large models. Finally, we demonstrated the portability of our method across different hardware environments, specifically on iPhone 13 Pro with an A15 Bionic CPU, a baremetal with an Intel Xeon 5218 Gold CPU, and a workstation with an NVIDIA A100 80GB GPU.

We introduced the three HQ variants, HQ1, HQ2, and HQ3, depending on the SQNR signal passing through the networks. The HQ2 method, which does not mix the signals of a quantized network with the baseline reference model, favors dynamic quantization over static quantization for linear layers. As a result, we recommend avoiding the HQ2 method in favor of dynamic quantization or the HQ1 and HQ3 variants.

On average, for the ViT family, we achieved 1.23, 1.13, and 1.68 speedups with HQ1 compared to dynamic quantization for an A15 CPU, an Intel Xeon 5218 Gold CPU, and an NVIDIA A100 GPU, respectively. For HQ3, we obtained an average speedup of 1.28, 1.15, and 1.68, respectively. Both HQ1 and HQ3 improved the top-1 accuracy compared to static quantization on the ImageNet1K validation dataset. For the ViT family, we recommend adopting the HQ1 method over HQ3.

Conforming to our previous findings, the DeiT and Swin Transformer models were robust to static quantization. As these families incurred the least quantization error after applying the static scheme. Within these models, we improved the top-1 accuracy compared to static quantization. Nevertheless, for the Swin Transformer, we observed that all methods favored static quantization over dynamic quantization for all models at almost every layer. As a result, we recommend that the Swin Transformer utilize static quantization with FP skip connections. As for DeiT, we observed for HQ1 an average speedup of 1.29, 1.20, and 2.02 compared to the dynamic quantization for the mobile A15 CPU, Intel Xeon 5218 Gold CPU, and an NVIDIA A100 GPU, respectively. For HQ3, we observed average speedups with respect to dynamic quantization of 1.21, 1.15, and 1.66 for the DeiT models. We recommend using static quantization, HQ1, or HQ3 for the DeiT models, depending on the required performance, target hardware, and use case.

Lastly, our method improved the top-1 accuracy of the DeiT3 family models.

We noted 1%, 4%, and 11.5% for small, base, and large variants over static quantization on the ImageNet1K validation dataset. Moreover, we recorded HQ1 average speedups of 1.33, 1.19, and 1.72 for the mobile A15 CPU, Intel Xeon 5218 Gold CPU, and an NVIDIA A100 GPU, respectively, while for HQ3, we obtained average speedups of 1.29, 1.18, and 1.69. We recommend the usage of HQ1 as it offers the best tradeoff concerning speed and accuracy. However, it was susceptible to high top-1 accuracy variability in DeiT3-B/16/224 and DeiT3-L/16/224. For more stable top-1 accuracy, we recommend the HQ3 method at the cost of top-1 accuracy and latency.

Chapter 5

Efficient GPU Kernels for Mixed-Precision Vision Transformers

5.1 Introduction

In the previous two chapters, we demonstrated two mixed-precision quantization schemes for ViT models: partial quantization and HQ. We also showed the latency improvements of HQ-quantized models compared to dynamic quantization using dedicated libraries and custom GPU kernels. Moreover, as mentioned in Chapter Section 2.6, many deep learning frameworks are dedicated to training, and a couple are dedicated to efficient inference of LLMs and DL models. Unfortunately, a gap exists between model training and GPU quantization inference in PyTorch. In this chapter, we propose a QAttn (Quantized attention, pronounced like katana), an open-source library that contains GPU-optimized quantized kernels for matrix multiplication and mixed-precision attention that utilizes Tensor core INT8 acceleration. These kernels comprise up to 99% of the operations (OPs) within the ViT-L/16/224 and take up to 96.34% of the time to process the input images, depending on the batch size, as shown in Table 5.1. Our framework integrates tightly with PyTorch FX and PyTorch 2.0 quantization workflows. As a result, it can be utilized as a replacement for CPU quantization. Moreover, the QAttn framework is available on GitHub¹.

¹<https://github.com/IBM/qattn>

In doing so, we make the following specific contributions:

- We develop QAttn, a Python framework with efficient GPU implementations of the linear layer and attention. Our integration with the quantization workflow in torch.fx [147] and torch.compile [148] allows in-place replacement of PyTorch modules.
- We evaluate the performance of the general matrix multiplication (GEMM) with PyTorch, TVM, TensorRT, and our proposed kernels over various shapes of linear layers of ViTs. The experimental results show that our implementation is competitive with closed-source TensorRT for statically quantized matrix multiplication. Moreover, it outperforms the TensorRT implementation and achieves over 460 TOP/s on certain kernels.
- We compare the throughput performance of mixed-precision attention with PyTorch memory efficient attention, the Triton FP16 reference, and FlashAttention2. Our mixed-precision attention outperforms the FP16 baseline implementation in Triton by up to 10.64% on average.
- We verify the numerical stability of our quantized and mixed-precision kernels on the ImageNet1K [22] validation dataset across different ViTs. The experiments with ViT-L models show less than 1% quantization error, with 75% weight reduction and up to 7x speedup compared to FP32.
- We apply per-channel dynamic and static quantization to the image encoder of the Segment Anything Model (SAM) [60]. We achieve over 5x more images processed per second for the base and large variants without mIOU (Equation (2.4)) drop over the COCO2017 [23] validation set. For the huge model, we achieve over 6x more images per second without any error for dynamic quantization, while for static, the mIOU dropped only by 2.66%.

Table 5.1: OPs by percentage and execution time for ViT-L/16/224. Number of layers, OPs, and latency are provided as a percentage [%] of the ViT model. We provide execution time for batch size $b = 1$ and $b = 128$.

Type	# Layers	OPs	$b = 1$	$b = 128$
Linear	40	96.43	61.87	87.16
Attention	10	3.09	12.51	9.18
Conv2D	< 1	0.25	0.51	0.27
Add	20	0.16	7.67	1.07
GELU	10	0.13	4.89	1.31
LayerNorm	20	0.08	12.52	1.01

5.2 Motivation

Modern DL models are usually trained using FP16/BFloat16 for storing weights and FP32 for calculating and storing gradients. As a result, the obvious choice for running the inference on accelerators is to use the same precision that was used during training. Nonetheless, modern accelerators are also equipped with dedicated cores for integer precision, e.g., NVIDIA Ampere generation supports INT8, INT4, and INT1 Tensor Core accelerated matrix multiplication. With INT8, the theoretical throughput is expected to be twice that of half precision on NVIDIA A100 GPU [46]. Moreover, INT8 consumes two times less memory for loading model weights and intermediate activations and requires significantly less energy [13, 149].

NVIDIA provides several libraries that make efficient use of its accelerators with Tensor cores. Among these, cuTLASS [150] is a CUDA C++ template library for efficient mixed-precision GEMM operation. While it is highly optimized for Tensor cores, it is not currently integrated with frameworks like PyTorch or JAX. Nonetheless, it is utilized by the TensorRT-LLM [116] framework, which focuses on mixed-precision, including quantized inference of LLMs. Unfortunately, TensorRT-LLM has limited support for CV compared to its predecessor, Faster-Transformer [117], which is no longer actively maintained. Lastly, TensorRT is a general-purpose DL inference framework to accelerate the models on NVIDIA GPUs and Jetson platforms. While it provides open-source frameworks to port the TensorFlow, PyTorch, or ONNX models, the core library and kernel implementations are closed-source.

The open-source framework *bitsandbytes* is designed for efficient training using mixed-precision data types [95]. The authors developed custom CUDA and Triton kernels for efficient INT8 and sub-byte types such as FP4 and normalized float 4-bit [93]. Those kernels enable efficient training of the models as the states of the optimizers and the weights are quantized [151]. Similarly to the TensorRT-LLM, the main target is LLMs.

In PyTorch, there are three options to quantize the DL model:

- The first includes rewriting the DL model’s code to introduce quantization primitives. Unfortunately, this does not utilize any graph capture mechanism and is not generalizable to various models.
- The second utilizes the torch.fx graph capture to extract the nodes and operations. On these primitives, we can easily insert observers into the DL model’s graph and replace the FP operations with their quantized counterparts. However, it has a limitation in that it cannot capture control graphs in the model’s code.

- The third uses the latest graph capture mechanism, `torchdynamo` - also referred to as a `torch.compile` workflow. `Torchdynamo` uses the CPython Frame Evaluation API to capture the model’s graphs. As a result, it can capture the control graphs within the model’s codebase. It uses the same notation as `torch.fx` to describe the model’s graphs with nodes and operations. We can write our custom torch backend with the captured model graph to replace the PyTorch operations with our custom kernels.

PyTorch natively supports the CPU-quantized kernels via FBGEMM [145] without support for GPU-accelerated kernels. We can extend the PyTorch with `TorchTensorRT` an open-source framework to lower PyTorch models to `TensorRT` but at the cost of flexibility and closed-source GPU kernels.

In summary, in this chapter, we describe an open-source framework `QAttn` that extends the PyTorch via `torch.fx` and `torch.compile` quantization workflows. We believe this framework will enable researchers to conduct studies more efficiently on hardware. Moreover, we propose static and dynamic mixed-precision attention implementation.

The rest of the chapter is structured as follows: in Section 5.3, we describe the `Triton` language and compiler, with an introduction to its features. In Section 5.4, we explain and present the implementation of quantized matrix multiplication in `Triton`, followed, in Section 5.5 our proposed mixed-precision attention and its implementation in `Triton`. These two kernels comprise over 99% of the OPs present in the ViT architecture. Next, in Section 5.6 we introduce `QAttn` an open source package that integrates with the PyTorch quantization workflows. This framework grants researchers and practitioners the possibility to evaluate and run their quantized DL models accelerated by GPUs. In Section 5.7, we define our experimental setup to validate quantized kernels and `QAttn`. Finally, in Section 5.8, we present our results starting in Section 5.8.1 with the quantized matrix multiplication kernel. Then, in Section 5.8.2, we show the results of the mixed-precision attention kernel. Section 5.8.3 examines the results obtained by running end-to-end tests evaluated on the ImageNet1K validation dataset. In Section 5.8.4. we present the results obtained on the instance segmentation task on the COCO2017 validation dataset. Lastly, in Section 5.9 we conclude and summarise our work on the implemented quantized kernels and `QAttn`.

5.3 Triton

`Triton` is a domain-specific language (DSL) that uses MLIR LLVM compiler for high-performance deep neural network operations. It provides a Python frontend API with just-in-time (JIT) compilation integrated with PyTorch [131]. `Triton`

offers an autotuning API for evaluating multiple iterations to find the fastest kernel. We explain in detail the autotuning feature in Section 5.4 in the context of matrix multiplication kernel. Moreover, it provides an option to define heuristics that reduce the search space. The Python DSL is translated to the Triton intermediate representation (TritonIR), which is later translated to MLIR LLVM and compiled into the device code.

As a result, Triton provides a high-level language that allows researchers and developers to program with higher productivity than low-level languages like CUDA. Moreover, it helps them optimize the memory transfers and manage the L2 cache in NVIDIA GPUs. Primarily created for the CUDA-enabled devices, other hardware vendors are adopting it.

5.4 Quantized Matrix Multiplication

In Equations (2.15) and (2.18) we defined the static and dynamic quantized matrix multiplication operations. While in Figures 5.1 and 5.2 we provide an implementation of static quantized matrix multiplication. Figure 5.1 offers the kernel signature with autotuning and heuristics annotation. Figure 5.2 displays an implementation of the static variant of quantized matrix multiplication implemented in the Triton language. In lines 33-34 and 1-2 of Figures 5.1 and 5.2 respectively, we define the kernel by using *@triton.jit* annotation with following input parameters:

- A, B, C: Pointers to the input matrices A and B and output matrix C.
- M, N, K: Dimensions of the matrices A, B, and C.
- stride_am, stride_ak, stride_bk, stride_bn, stride_cm, stride_cn: Distance in the memory from one column or row in the given matrix to access the next element.
- a_scale_ptr, b_scale_ptr, c_scale_ptr: Pointers to the scale factors of A, B, and C matrices.
- BLOCK_M, BLOCK_N, BLOCK_K: Marked as compile time constants with *tl.constexpr*. The size of the chunks of matrix multiplications along the M, N, and K dimensions for parallel processing.
- GROUP_M: Number of block groups along the M dimension.
- EVEN_K: Boolean value to check if K dimension is multiple of BLOCK_K.

In Figure 5.1, we provide an overview of autotuning applied to a quantized matrix multiplication kernel. First, we define a helper function in line 4 that

generates the search space of configurations for an NVIDIA A100 GPU. In lines 9-12, we append a *triton.Config* object with defined block sizes for M, N, and K. Moreover, we define a number of stages and warps. A number of stages refers to the pipeline stages in the NVIDIA GPU. With a larger number of stages, we can hide latency by pipelining the data transfer and computation. The number of warps corresponds to spawned GPU warps that contain the thread blocks that execute the GPU kernel.

During the autotuning, Triton will test every configuration to measure the execution time and select the best configuration. To avoid the problems of cold-start and long execution, we can add functions that will prune the configs and estimate the time based on the number of TFLOP/s. In this way, the search space is significantly reduced. Moreover, we can provide heuristics that substitute the parameters depending on other parameters passed to the kernel. Triton also provides an option to manually specify the *tl.constexpr* parameters to avoid autotuning.

Figure 5.2 provides a simplified version for brevity that assumes that the K dimension is even (in implementation and Figure 5.1 we consider non-even matrices). However, to tackle these ViT cases when this is not true in the case of the ViTs, we pad the blocks with 0, which incurs tile quantization. For example, for a block of size 256×257 , we would have to spawn additional thread block tiles to 256×384 to cover an uneven shape of the matrix. Moreover, we fused the bias add operation in our kernel implementation.

Triton abstracts the CUDA semantics related to grids, blocks, and threads - it introduces the *tl.program_id* function that returns the id of the current program along the given dimension. In Triton we can launch the programs alongside three dimensions. First, in lines 4-7, we get the thread block position in the 2D grid, and then we calculate how many blocks are needed to cover the BLOCK_M and BLOCK_N positions. Next, in lines 9-13 we calculate the thread position and the sizes of the groups. This is followed by calculating the memory offsets *ram* and *rbn* that are aligned and continuous within the memory block. Finally, in lines 23 and 24, we move the pointers for A and B matrices to the starting position for the given thread. Finally, in lines 23 and 24, we move the pointers of A and B matrices to the starting position for the given thread. We initialize our accumulator in line 25 of size BLOCK_M and BLOCK_N and datatype INT32 to avoid overflows during computation.

The computation of the matrix multiplication is outlined in lines 27-32. We define the inner loop over the K dimension of the matrices. We load the matrices A and B into local variables within the for loop. If the K dimension is uneven, we need to consider utilizing the masking of the load operation not to load adjacent memory. Moreover, we pad the tile with zeros to match the BLOCK_K size. Then,

we add the results of the dot product of these matrices into our accumulator. *tl.dot* function is the intrinsic function in the Triton language and, depending on the input data type, shape, and output parameter, translates to the device code that will execute on Tensor cores. After computation, the pointers A and B are advanced to the next memory block along the K dimension.

As the accumulator is of INT32 data type and we expect the output matrix C to be of INT8 type, we first need to load the scaling factors (lines 35-37). Then, we cast the accumulator to FP32 and multiply it by the requantization scale. Lastly, we round it to the nearest integer and cast it to INT8. This requantization step showcases the scalar variant. However, it is possible to load per-channel scalar factors. In this manner, we would have to utilize indices to calculate the proper scaling factor for the given thread block.

Finally, we recalculate the offsets of the C pointer to store the results in the correct memory location. We also compute the memory mask so as not to write outside of the result memory layout. Similar to CUDA, we need to make sure that the passed output pointer is of the same type as the accumulator to avoid writing to the wrong memory location.

In the case of the dynamic quantization, we would load the matrix A of FP data type (FP32/FP16/BFloat16) and quantize the block chunk before performing the dot product. Contrary to the static quantization, we do not have a precomputed scaling factor for matrix A. Before executing the kernel, we compute it with the maximum absolute value of the matrix A as defined in Equation (2.11). Moreover, we would omit the *c_scale_ptr*, and the output of this kernel would be in FP.

```

1 import triton
2 import triton.language as tl
3
4 def int8_configs():
5     configs = []
6     n_stages_n_warps = [(s, w) for s, w in product(range(1, 9), [1, 2, 4, 8])]
7     for num_stages, num_warps in n_stages_n_warps:
8         for m, n, k in product([64, 128, 256], [64, 128, 256], [64, 128, 256]):
9             configs.append(triton.Config(
10                 {"BLOCK_M": m, "BLOCK_N": n, "BLOCK_K": k},
11                 num_stages=num_stages, num_warps=num_warps
12             ))
13     return configs
14
15 @triton.autotune(configs=int8_configs(), key=["M", "N", "K"])
16 @triton.heuristics({"EVEN_K": lambda args: args["K"] % args["BLOCK_K"] == 0})
17 @triton.jit
18 def _static_matmul_kernel(A, B, C, M, N, K, stride_am, stride_ak, stride_bk,
19     stride_bn, stride_cm, stride_cn, a_scale_ptr, b_scale_ptr, c_scale_ptr,
20     BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr,
21     GROUP_M: tl.constexpr, EVEN_K: tl.constexpr): ...

```

Figure 5.1: Triton autotune and heuristics decorators are used in the static matrix multiplication kernel. We provide various configurations on a number of stages and warps as well as block sizes that the autotuner tries to optimize over.

5. EFFICIENT GPU KERNELS FOR MIXED-PRECISION VISION TRANSFORMERS

```

1 @triton.jit
2 def _static_matmul_kernel(A, B, C, M, N, K, stride_am, stride_ak, stride_bk,
    stride_bn, stride_cm, stride_cn, a_scale_ptr, b_scale_ptr, c_scale_ptr,
    BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr,
    GROUP_M: tl.constexpr):
3     # Localize the grid and thread blocks
4     pid = tl.program_id(0)
5     pid_z = tl.program_id(1)
6     grid_m = tl.cdiv(M, BLOCK_M)
7     grid_n = tl.cdiv(N, BLOCK_N)
8     # Group the computational blocks
9     width = GROUP_M * grid_n
10    group_id = pid // width
11    group_size = min(grid_m - group_id * GROUP_M, GROUP_M)
12    pid_m = group_id * GROUP_M + (pid % group_size)
13    pid_n = (pid % width) // group_size
14
15    # Calculate the offsets
16    rm = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
17    rn = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
18    ram = tl.max_contiguous(tl.multiple_of(rm % M, BLOCK_M), BLOCK_M)
19    rbn = tl.max_contiguous(tl.multiple_of(rn % N, BLOCK_N), BLOCK_N)
20    rk = pid_z * BLOCK_K + tl.arange(0, BLOCK_K)
21
22    # Move the pointers to the correct memory offset
23    A = A + (ram[:, None] * stride_am + rk[None, :] * stride_ak)
24    B = B + (rk[:, None] * stride_bk + rbn[None, :] * stride_bn)
25    acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.int32)
26    # Matrix multiplication product over K dimension
27    for k in range(0, tl.cdiv(K, BLOCK_K)):
28        a = tl.load(A) # a is BLOCK_M x BLOCK_K
29        b = tl.load(B) # b is BLOCK_K x BLOCK_N
30        acc += tl.dot(a, b, allow_tf32=False, out_dtype=tl.int32)
31        A += BLOCK_K * stride_ak
32        B += BLOCK_K * stride_bk
33
34    # Load scaling factors
35    a_scale = tl.load(a_scale_ptr)
36    b_scale = tl.load(b_scale_ptr)
37    out_scale = tl.load(c_scale_ptr)
38
39    # Requantize step
40    out_scale = a_scale * b_scale * out_scale
41    acc = tl.math.llrint((acc.to(tl.float32) * out_scale)).to(tl.int8)
42
43    # Recalculate the offsets for storing the outputs
44    rm = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
45    rn = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
46    C = C + (rm[:, None] * stride_cm + rn[None, :] * stride_cn)
47    mask = (rm < M)[:, None] & (rn < N)[None, :]
48    tl.store(C, acc, mask=mask)

```

Figure 5.2: A simplified quantized static matrix multiplication kernel implemented in Triton. The matrix A is of size $M \times K$, matrix B is of size $K \times N$, while the output matrix C is of size $M \times N$. For brevity, we omit the bias parameter. Moreover, we for simplicity assume the K dimension to be even.

5.5 Mixed-Precision Attention

Attention is a pivotal operation within the transformer architecture. It allows the transformer to focus dynamically on the important patches or tokens within the sequence. As introduced in Section 2.3, the attention mechanism consists of two matrix multiplications with Softmax activation in between. The main bottleneck in the attention operation is the data movement from each matrix multiplication and Softmax. Therefore, FlashAttention2 was proposed, focusing on an NVIDIA A100 GPU with FP16/BFloat16 data types as inputs and outputs [39].

Based on the FlashAttention2 kernel, we propose quantized mixed-precision attention. Our method extends this operation to accept the inputs as INT8 and, in the case of static quantization, also the output. In Figure 5.3, we provide the kernel signature and configurations used to tune the kernel. Similarly to the matrix multiplication kernel, we define the search space over BLOCK_M and BLOCK_N, the number of stages, and the number of warps.

We perform autotuning of the kernel based on the sequence length (N_CTX), number of heads (H), and batch size (Z). Moreover, we add heuristics to check for even sequence length to add boundary checks for loading the matrices (for brevity, they were removed in Figures 5.3 and 5.4). Checks for even sequence length are required for ViTs, as this length is usually suspect for tile quantization. The kernel accepts the following parameters:

- Q, K, V, Out: Input matrices query, key, value and output matrix pointers.
- sm_scale: Softmax scale calculated as $1/\sqrt{d}$, where d is the head dimension.
- qkv_scale_ptr out_scale_ptr: Pointers to the quantization scaling factors.
- stride_*: Stride parameters for accessing the elements in matrices Q, K, V, and Out along Z, h , M, and K dimensions
- Z, h , N_CTX: Batch size, number of attention heads, and the sequence (context) length.
- BLOCK_M, BLOCK_DMODEL, BLOCK_N: Marked as compile time constants with *tl.constexpr*. The size of the chunks of the blocks of Q, K, and V matrices used

Similarly to the matrix multiplication kernel, we first calculate the memory offsets in the given thread block in lines 31-33 in Figure 5.3. Next, in lines 35-49, we initialize the block pointers using *tl.make_block_ptr* API. This function helps to avoid calculating masks for each of the pointers. Lastly, we initialize

accumulators: m_i will keep the maximum value for a given row, l_i accumulates the normalization factor for Softmax, while acc is an accumulator of the second dot product.

First, for static mixed-precision quantization, we need to run the calibration step and calculate the scale factors for query the (Q), key (K), value (V), and output matrices. We perform the first matrix multiplication in INT8 to ensure the numerical stability and dequantize it to FP32; see Figure 5.4 lines 10-19:

$$\bar{E}_1 = \frac{s_Q s_K}{\sqrt{d}} \mathcal{Q}(\mathcal{C}(\mathcal{Q}(Q), \mathcal{Q}(K))), \quad (5.1)$$

where s_q and s_k are scaling factors for Q and K matrices, the d represents the head size. Next, we calculate the attention weights with the Softmax function:

$$E_1 = \text{Softmax}(\bar{E}_1). \quad (5.2)$$

As we are performing the tiling computation, we compute the Softmax with scaling at lines 22-26 in Figure 5.4. Next, we load the V matrix in INT8 into the kernel and dequantize it to BFloat16 data type; see Figure 5.4 lines 28-29:

$$\hat{V} = \mathcal{Q}(V) \cdot s_V. \quad (5.3)$$

The second matrix multiplication is performed in FP arithmetic:

$$\text{Out} = E_2 = \mathcal{C}(E_1, \hat{V}). \quad (5.4)$$

Similarly, as the implementation is based on the tiling, we first scale the accumulator in the kernel and then perform the matrix multiplication at lines 30-31 Figure 5.4.

In the case of dynamic quantization, we return the output matrix Out in BFloat16, while in the static variant, we need to quantize it to INT8:

$$\mathcal{Q}(\text{Out}) = \lfloor \frac{\text{Out}}{s_O} \rfloor. \quad (5.5)$$

In the kernel, we perform multiplication instead of division during the quantization step; line 39 Figure 5.4. Before launching the kernel, we store the scaling factor as its inverse. We perform this operation because the GPU multiplication instruction requires less cycles than division.

```

1 import triton
2 import triton.language as tl
3
4
5 def _get_configs():
6     configs = []
7     for block_m in [64, 128, 256]:
8         for block_n in [32, 64, 128]:
9             for num_stages in [3, 4, 5, 6, 7, 8]:
10                 for num_warps in [4, 8]:
11                     configs.append(
12                         triton.Config(
13                             {"BLOCK_M": block_m, "BLOCK_N": block_n},
14                             num_warps=num_warps,
15                             num_stages=num_stages,
16                         )
17                     )
18     return configs
19
20
21 @triton.autotune(configs=_get_configs(), key=["N_CTX", "h", "Z"])
22 @triton.jit
23 def _static_attention_kernel(Q, K, V, sm_scale, qkv_scale_ptr, out_scale_ptr,
24                             Out, stride_qz, stride_qh, stride_qm, stride_qk, stride_kz,
25                             stride_kh, stride_kn, stride_kk, stride_vz, stride_vh,
26                             stride_vk, stride_vn, stride_oz, stride_oh,
27                             stride_om, stride_on, Z, h, N_CTX,
28                             BLOCK_M: tl.constexpr, BLOCK_DMODEL: tl.constexpr, BLOCK_N: tl.constexpr
29 ):
30     # Initialize program_ids and offsets
31     start_m, off_hz = tl.program_id(0), tl.program_id(1)
32     off_z, off_h = off_hz // h, off_hz % h
33     qvk_offset = off_z.to(tl.int64) * stride_qz + off_h.to(tl.int64) *
34         stride_qh
35     # Create block pointers for Q, K, and V
36     Q_block_ptr = tl.make_block_ptr(
37         base=Q + qvk_offset, shape=(N_CTX, BLOCK_DMODEL),
38         strides=(stride_qm, stride_qk), offsets=(start_m * BLOCK_M, 0),
39         block_shape=(BLOCK_M, BLOCK_DMODEL), order=(1, 0),
40     )
41     K_block_ptr = tl.make_block_ptr(
42         base=K + qvk_offset, shape=(BLOCK_DMODEL, N_CTX),
43         strides=(stride_kk, stride_kn), offsets=(0, 0),
44         block_shape=(BLOCK_DMODEL, BLOCK_N), order=(0, 1),
45     )
46     V_block_ptr = tl.make_block_ptr(
47         base=V + qvk_offset, shape=(N_CTX, BLOCK_DMODEL),
48         strides=(stride_vk, stride_vn), offsets=(0, 0),
49         block_shape=(BLOCK_N, BLOCK_DMODEL), order=(1, 0),
50     )
51     # Initialize accumulators
52     m_i = tl.zeros([BLOCK_M], dtype=tl.float32) - float("inf")
53     l_i = tl.zeros([BLOCK_M], dtype=tl.float32)
54     acc = tl.zeros([BLOCK_M, BLOCK_DMODEL], dtype=tl.float32)
55     ...

```

Figure 5.3: Mixed-precision attention kernel signature and autotuning options. This code block presents the kernel definition and initial phase of calculating the offsets and preparing the block pointers and accumulators for the kernel execution.

```

1 @triton.jit
2 def _static_attention_kernel(...):
3     ...
4     # Load qkv scale quantization parameter
5     qkv_scale = tl.load(qkv_scale_ptr)
6     qk_scale = qkv_scale * qkv_scale * sm_scale * 1.44269504
7
8     # Load q: it will stay in SRAM throughout
9     q = tl.load(Q_block_ptr, boundary_check=(0,), padding_option="zero")
10    for start_n in range(0, N_CTX, BLOCK_N):
11        # compiler hint
12        start_n = tl.multiple_of(start_n, BLOCK_N)
13        # Load k
14        k = tl.load(K_block_ptr, boundary_check=(1,), padding_option="zero")
15        # Compute qk
16        qk = tl.zeros([BLOCK_M, BLOCK_N], dtype=tl.int32)
17        qk += tl.dot(q, k, allow_tf32=False, out_dtype=tl.int32)
18        # Dequantize qk to FP32
19        qk_fp32 = qk * qk_scale
20
21        # Compute scaling constant
22        m_ij = tl.maximum(m_i, tl.max(qk_fp32, 1))
23        p = tl.math.exp2(qk_fp32 - m_ij[:, None])
24        # Scale and update acc
25        alpha = tl.math.exp2(m_i - m_ij)
26        m_i = m_ij
27        # Load v
28        v = tl.load(V_block_ptr, boundary_check=(0,), padding_option="zero")
29        v = (v * qkv_scale).to(tl.bfloat16)
30        acc *= alpha[:, None]
31        acc += tl.dot(p.to(tl.bfloat16), v, allow_tf32=True)
32        l_i = l_i * alpha + tl.sum(p, 1)
33        # Update pointers
34        K_block_ptr = tl.advance(K_block_ptr, (0, BLOCK_N))
35        V_block_ptr = tl.advance(V_block_ptr, (BLOCK_N, 0))
36
37        # Scale and quantize the output
38        out_scale = tl.load(out_scale_ptr)
39        acc = tl.math.lrint(acc / (l_i[:, None] * out_scale)).to(tl.int8)
40        # Store the results
41        O_block_ptr = tl.make_block_ptr(
42            base=Out + qvk_offset,
43            shape=(N_CTX, BLOCK_DMODEL),
44            strides=(stride_om, stride_on),
45            offsets=(start_m * BLOCK_M, 0),
46            block_shape=(BLOCK_M, BLOCK_DMODEL),
47            order=(1, 0),
48        )
49        tl.store(O_block_ptr, acc, boundary_check=(0,))

```

Figure 5.4: Mixed-precision attention kernel implementation. This code block presents the mixed-precision attention computation instructions. Here, we assume that the matrices might not be even in context length dimension. We check boundaries when we load and store the matrices to avoid wrong memory access.

5.6 QAttn

We designed QAttn to be compatible with the PyTorch quantization workflows: `torch.fx` and `torch.compile`. It targets inference and PTQ scenarios. Inside QAttn, we implemented the GPU kernels to efficiently perform the INT8 static and dynamic quantized matrix multiplication used in the linear layers and mixed-precision attention.

We extended `torch.fx` by adding the backend configuration that is being imported in line 6 in Figure 5.5. Moreover, in QAttn, we provided the default quantization config (`qconfig`) that defines the observers, data types, and quantization scheme. It is possible to fetch a configuration object for static or dynamic quantization as well as for scalar and per-channel quantization. Next, lines 13-24 follow the `torch.fx` quantization workflow. In the case of dynamic quantization, the calibration step can be omitted as dynamic quantization does not require activation observers. Finally, we call the `convert` function of the QAttn framework, which lowers marked and supported operators to the equivalent quantized modules and functions that utilize the GPU kernels implemented in Triton.

```

1 import torch
2 import torch.nn.functional as F
3 from torch.ao.quantization import quantize_fx, QConfigMapping
4 import timm
5 import qattn
6 from qattn.backends_config.qattn import get_qattn_backend_config
7
8 # Create model
9 model = timm.create_model("vit_large_patch16_224", pretrained=True).eval()
10 # Get default qconfig
11 qconfig = qattn.get_default_qconfig(is_dynamic=False, per_channel=False)
12 # Define QConfigMapping
13 mapping = QConfigMapping().set_global(None).set_object_type(torch.nn.Linear,
14                       qconfig).set_object_type(F.scaled_dot_product_attention, qconfig)
15 # Example input, can be arbitrary.
16 example_input = torch.randn(1, 3, 224, 224)
17 model = quantize_fx.prepare_fx(
18     model, example_inputs=(example_input,),
19     qconfig_mapping=mapping, backend_config=get_qattn_backend_config(),
20 ).to("cuda")
21 # Calibrate the model
22 for _ in range(5):
23     with torch.inference_mode():
24         x = torch.randn(1, 3, 224, 224).to("cuda")
25         _ = model(x)
26 # Convert the model
27 model = qattn.convert(model)
28 _ = model(example_input.to("cuda"))

```

Figure 5.5: An example of using the QAttn with `torch.fx` quantization workflow on the ViT-L/16/224 model.

PyTorch 2.0 introduced the `torch.compile` quantization workflow, which uses `torchdynamo` for graph capture. As a result, this kind of workflow captures more dynamic models than the `torch.fx` graph capture. We extend this workflow by adding a `QAttnQuantizer` object that annotates the model nodes for quantization. In line 12 in Figure 5.6, we instruct the quantizer to annotate the linear modules in the ViT-L/16/224 with our specified default static quantization configuration. Similarly to `torch.fx` workflow, we support dynamic quantization and per-channel quantization.

Lines 21-28 present the standard `torch.compile` quantization workflow utilizing the `torchdynamo` graph capture mechanism. After calling the `convert_pt2e` function, we get the model graph that consists of base ATen operations and simulated quantization nodes. ATen is a tensor library that PyTorch builds upon and defines operations primitives used in DL. In line 30, we mark the `converted_model` to be compiled by our backend implementation. In line 31 our backend is invoked by passing the first input to the DL model. Inside the backend, we replace quantization nodes and ATen operations with our implemented quantized GPU kernels.

```

1 import timm
2 import torch
3 from torch._export import capture_pre_autograd_graph
4 from torch.ao.quantization.quantize_pt2e import prepare_pt2e, convert_pt2e
5 from qattn.pt2e.quantizer import (
6     QAttnQuantizer,
7     get_default_qattn_quantization_config,
8 )
9
10 # Initialize quantizer
11 quantizer = QAttnQuantizer()
12 quantizer.set_module_type(
13     torch.nn.Linear,
14     get_default_qattn_quantization_config(is_dynamic=False),
15 )
16
17 # Initialize model and sample
18 model = timm.create_model("vit_large_patch16_224", pretrained=True)
19 model = model.to("cuda").eval()
20 sample = torch.randn((1, 3, 224, 224), device="cuda:0")
21
22 # Export and prepare the model
23 exported_model = capture_pre_autograd_graph(model, (sample,))
24 prepared_model = prepare_pt2e(exported_model, quantizer).to(device="cuda")
25
26 # Calibrate for static quantization
27 for _ in range(5):
28     with torch.inference_mode():
29         x = torch.randn(1, 3, 224, 224).to("cuda")
30         _ = prepared_model(x)
31 converted_model = convert_pt2e(prepared_model, fold_quantize=True)
32
33 # Invoke lowering via torch compile
34 model = torch.compile(converted_model, backend="qattn")
35 _ = model(sample)

```

Figure 5.6: An example of using the `QAttn` with `torch.compile` quantization workflow on the ViT-L/16/224 model.

In both workflows we get statically or dynamically quantized ViT, depending on the configuration. However, since QAttn currently supports only a limited number of GPU kernels, the remaining operations are performed in the FP32, FP16, or BFloat16. To mitigate this, we insert quantization and dequantization operators before and after the quantized kernels as needed. Nevertheless, the quantized model can be used in inference at any point where the reference model was used before.

5.7 Experimental Setup

In our setup, we used a workstation equipped with an NVIDIA A100 GPU with an 80GB HBM2e RAM. We installed PyTorch 2.2.1 [122, 148], Triton 2.2.0 [131], TensorRT 8.6.1 [115], and TVM 0.13.0 [128] on this workstation. First, we focus on the performance of each GPU kernel, expressed in tera operations per second (TOP/s). We note that the FP arithmetic performance is expressed in tera floating point operations per second (TFLOP/s). However, to unify the metric with quantized kernels that perform integer operations, we will refer to TFLOP/s as TOP/s.

We identified different linear layers and attention shapes across ViTs [8] and evaluated the raw performance of these operations with varying batch sizes $b = \{1; 32; 64; 128; 256; 512; 1024; 2048\}$. We compared our static (INT8) and dynamic (INT8-D) quantized matrix multiplication kernel with FP32 GPU PyTorch, a custom INT8 GPU TVM kernel, and static quantized INT8 TensorRT obtained by lowering the linear layer with the Torch-TensorRT package [152]. For each kernel, we ran 100 warm-up steps, after which we evaluated the kernel function 1000 times with L2 cache flushed between the calls.

The performance of quantized mixed-precision attention was evaluated against PyTorch FP16 memory efficient attention, fused attention implemented in Triton using FP16, and FlashAttention2 (v.2.5.6) [39]. We fixed batch size $b = 512$, head dimension $d = 64$, number of heads $h = \{12, 16\}$. The head dimension d and the number of heads H are ViT-B and ViT-L default values. We set the variable sequence length $N = \{197, 256, 512, 577, 785, 1024, 2048, 4096, 8192\}$. Sequence lengths of 197, 577, and 785 are used in the ViT architecture, depending on the model patch size P and input image size. Likewise, for the matrix multiplication kernel, we ran 100 warm-up steps, after which we evaluated the kernel function 1000 times with L2 cache flushed between the calls.

Next, we evaluated the end-to-end performance of the ViT models on the ImageNet1K [22] validation dataset over five runs. We report the top-1 accuracy of the FP32, static quantized linear layer (INT8), static quantized linear layer

with static mixed-precision attention (INT8+A), dynamic quantized linear layer (INT8-D), dynamic quantized linear layer with dynamic mixed-precision attention (INT8-D+A), PTQ4ViT [98], and Zhang et al [101]. We calibrated the static quantized models on the 2000 samples from the ImageNet1K training dataset with absolute maximum observer (Equation (2.11)). Additionally, we report the throughput speedup over FP32 with our static quantized kernels. Checkpoints of ViTs were fetched from the timm package. We consider models of sizes small (ViT-S), base (ViT-B), and large (ViT-L) with patch size $P = \{16; 32\}$ and input image size of 224×224 or 384×384 . For the ViT evaluation, we used the QAttn torch.fx workflow.

Finally, we extend our tests to instance segmentation on the COCO2017 validation dataset [23]. We report the mean intersection over union (mIOU) and images processed per second metrics for this task. We evaluated our QAttn torch.compile workflow with per-channel static and dynamic quantization on the Segment Anything Model (SAM) [60]. We consider the quantization of the linear layers of the image encoder model within the SAM. The image encoder comes in different sizes, based on the number of parameters, which we classify a shSAM-B, SAM-L, and SAM-H. Similarly, we calibrated each image encoder on 2000 random samples from the COCO2017 training dataset for static quantization with absolute maximum observer (Equation (2.11)). We compared our static (INT8) and dynamic (INT8-D) implementations with the SAM FP32 base implementation² and Segment Anything Fast dynamic quantization (INT8-D*)³.

5.8 Results

5.8.1 Matrix Multiplication Kernel

We evaluate the kernels representing the shapes of the ViT-S/32/224 fully connected layers in the MLP block. The comparison between PyTorch FP32, INT8 TVM, our static and dynamic quantized INT8 Triton, and INT8 TensorRT are shown in Figures 5.7 and 5.8. The first fully connected layer has the shape of $M = 50$, $K = 384$, and $N = 1536$, while the second fully connected layer has the dimensions of $M = 50$, $K = 1536$, and $N = 384$. For both kernels and batch size $b = 1$, the performance achieved by all kernels was below 1 TOP/s except for the TensorRT implementation, which achieved 1.01 TOP/s. Next, we observe a plateau of the TVM INT8 kernel at around 17 TOP/s at a batch size equal to 64. This is the same performance as PyTorch’s CUDA FP32 kernel, which does not use the Tensor

²<https://github.com/facebookresearch/segment-anything>

³<https://github.com/pytorch-labs/segment-anything-fast>

cores. Unfortunately, TVM IR cannot use the accelerator’s specific instructions to achieve higher throughput.

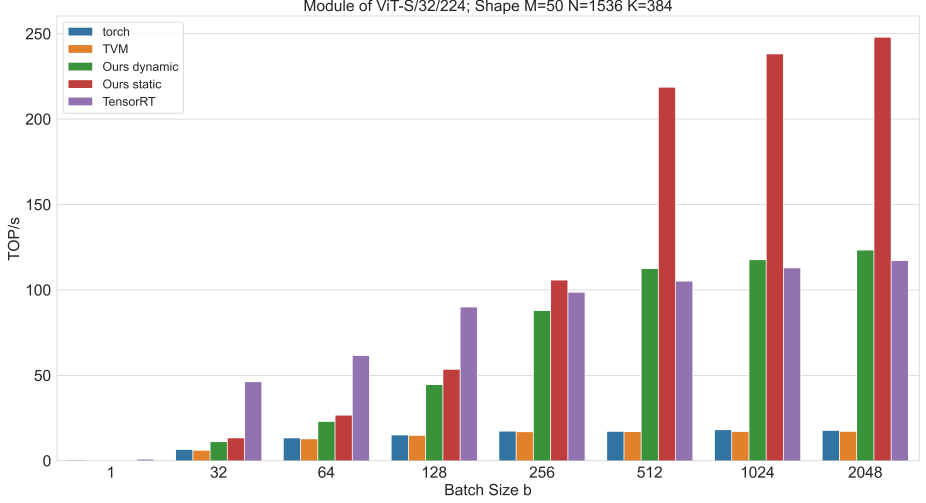


Figure 5.7: Matrix multiplication with dimensions M , N and K over different batch sizes of ViT-S/32/224 linear layer. Reference Pytorch runs in FP32 and TensorRT in INT8. Our dynamic version accepts FP16 inputs with weights in INT8; our static implementations Triton and TVM run in INT8.

In the first fully connected layer of the MLP block in ViT-S/32/224, the TensorRT INT8 implementation outperformed our static Triton implementation at batch sizes 1, 32, 64, and 128. At batch size $b = 128$, it reached 90.05 TOP/s, while our kernel reached 53.56 TOP/s. Starting with a batch size $b = 256$, our static matrix multiplication was faster than the TensorRT implementation. Moreover, at the batch size $b = 512$, our dynamically quantized matrix multiplication achieved 123.13 TOP/s while TensorRT attained 105.20 TOP/s. The TensorRT implementation had a maximum throughput of 117.29 TOP/s, while our static Triton kernel achieved more than twice the performance with 247.97 TOP/s.

In the second fully connected layer of the MLP block in ViT-S/32/224, we observed that our dynamically quantized matrix multiplication kernel did not outperform the one in TensorRT, whose performance was capped at 87.10 TOP/s. Furthermore, the TensorRT routine outperformed our statically quantized kernel at batch size $b = 256$ with 110.47 TOP/s compared to 100.41 TOP/s. However, TensorRT’s performance for this module was capped at 135.97 TOP/s compared

to our statically quantized matrix multiplication kernel which delivered 304.11 TOP/s.

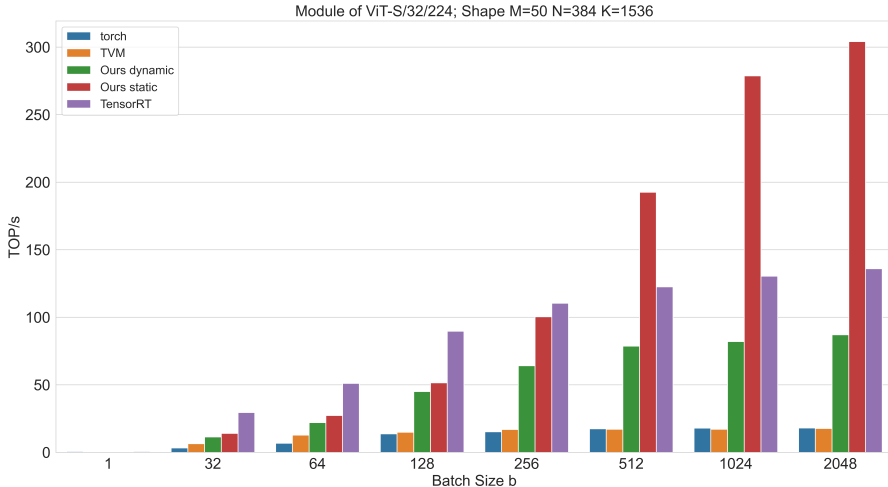


Figure 5.8: Matrix multiplication with dimensions M, N and K over different batch sizes of ViT-S/32/224 linear layer. Reference Pytorch runs in FP32 and TensorRT in INT8. Our dynamic version accepts FP16 inputs with weights in INT8; our static implementations Triton and TVM run in INT8.

Figure 5.9 shows the performance of the second fully connected layer in the ViT-B/16/224. As in the previous examples, we observed a limited performance of the PyTorch FP32 CUDA kernel and the TVM INT8 kernel. The performance of our dynamically quantized kernel reached 123.51 TOP/s with a batch size $b = 512$ and did not improve much with a larger batch size. In the first fully connected layer, the dynamically quantized layer reached up to 172.28 TOP/s with a batch size $b = 2048$.

Our statically quantized kernel outperformed TensorRT at batch size $b = 32$ with 195.58 TOP/s compared to 185.38 TOP/s. Furthermore, at batch size $b = 64$, our kernel achieved 404.46 TOP/s compared to TensorRT’s 201.26 TOP/s. Both kernels saturated at dimensions $b = 512$, $M = 197$, $N = 768$, and $K = 3072$, with our implementation reaching 460.25 TOP/s while TensorRT reached 224.27 TOP/s. Similar observations were made for a matrix multiplication with dimensions $M = 197$, $N = 3072$, and $K = 768$. The saturated kernels at batch size $b = 512$ with our kernel throughput reached 340 TOP/s while TensorRT reached 225 TOP/s.

In ViT-L/16/224, the dimensions of the second fully connected layer are $M=197$, $N=1024$, and $K=4096$; see Figure 5.10. Repeating the previous results, the PyTorch

CUDA FP32 and TVM INT8 kernel peaked at 18 TOP/s due to the lack of utilizing Tensor cores in the NVIDIA GPU. The dynamically quantized matrix multiplication kernel reached 135.54 TOP/s and 135.81 TOP/s at batch sizes $b = 1024$ and $b = 2048$, respectively. Likewise, we found that the dynamically quantized matrix multiplication with dimensions $M = 197$, $N = 4096$, and $K = 1024$ achieved a throughput of up to 165 TOP/s for this model.

For the batch size $b = 1$, TensorRT achieved 33.22 TOP/s compared to our statically quantized matrix multiplication kernel at 12.28 TOP/s. Nevertheless, with a larger batch size $b = 32$, we achieved 352.94 TOP/s, that is over 51% faster than TensorRT’s 232.66 TOP/s. Moreover, our kernel slightly improved the throughput with increasing batch size, reaching 450.85 TOP/s and 481.48 TOP/s for batch sizes $b = 128$ and $b = 2048$, respectively. At the same time, TensorRT capped the throughput at 260.32 TOP/s at batch size $b = 128$ and reached a peak performance of 266.45 TOP/s at batch size $b = 2048$.

Lastly, with increased patch and input image size in the fully connected layers of ViT-L/32/384, the M dimension equals 145, while the N and K stay the same as in ViT-L/16/224 for both matrix multiplication kernels. We observed the same pattern with PyTorch FP32, TVM INT8, and our dynamically quantized matrix multiplication kernel. We achieved peak throughput at batch size $b = 2048$ equal to 135.45 TOP/s and 165.92 TOP/s for the first and second fully connected layers,

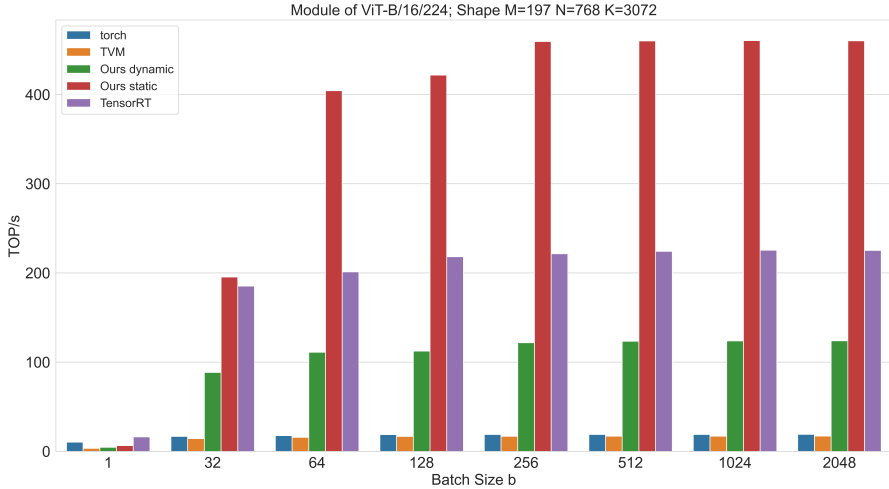


Figure 5.9: Matrix multiplication with dimensions M , N and K over different batch sizes of ViT-B/16/224 linear layer. Reference Pytorch runs in FP32 and TensorRT in INT8. Our dynamic version accepts FP16 inputs with weights in INT8; our static implementations Triton and TVM run in INT8.

5. EFFICIENT GPU KERNELS FOR MIXED-PRECISION VISION TRANSFORMERS

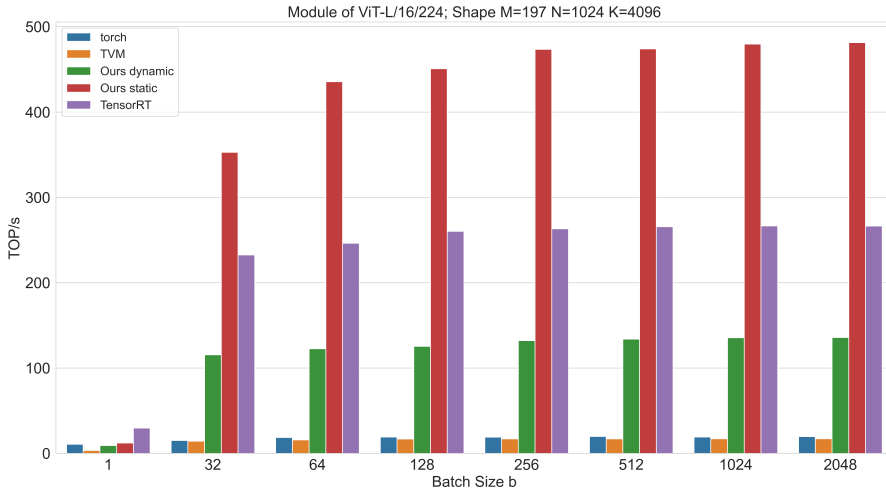


Figure 5.10: Matrix multiplication with dimensions M, N and K over different batch sizes of ViT-L/32/224 linear layer. Reference Pytorch runs in FP32 and TensorRT in INT8. Our dynamic version accepts FP16 inputs with weights in INT8; our static implementations Triton and TVM run in INT8.

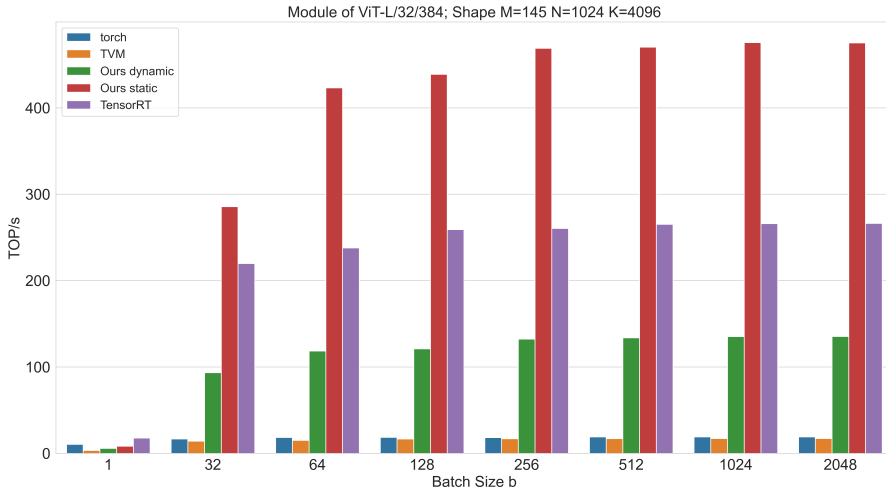


Figure 5.11: Matrix multiplication with dimensions M, N and K over different batch sizes of ViT-L/32/384 linear layer. Reference Pytorch runs in FP32 and TensorRT in INT8. Our dynamic version accepts FP16 inputs with weights in INT8; our static implementations Triton and TVM run in INT8.

respectively; see Figure 5.11.

TensorRT static quantized kernel for matrix multiplication only outperformed our static Triton kernel for batch size $b = 1$ with 17.81 TOP/s compared to 8.43 TOP/s. Nevertheless, with batch size $b = 32$, we achieved 30% faster throughput than TensorRT - 285.70 TOP/s compared to 219.81 TOP/s. The TensorRT plateaued at $b = 128$ for the first fully connected layer with 235.02 TOP/s, while the kernel with dimensions of the second fully connected layer achieved peak throughput at batch size $b = 2048$ yielding 266.31 TOP/s. At the same time, we achieved the peak throughput with our statically quantized matrix multiplication kernel at 377.58 TOP/s and 475.57 TOP/s at batch size $b = 1024$ for the first and second fully connected layers in the ViT-L/32/384, respectively, which makes our kernel implementation 60% and 78% faster than TensorRT, respectively.

5.8.2 Attention Kernel

Figure 5.12 shows the throughput of different attention kernels with twelve heads ($h = 12$). We note that our static mixed-precision Triton kernel outperformed the FlashAttention2 CUDA kernel at sequence lengths equal (N) to 197, 577, and 785. These lengths correspond to the ones used in the ViT. Our static mixed-precision kernel achieved 89, 129.5, and 132.7 TOP/s compared to FlashAttention2 - 78.7, 128.8, and 128.3 TOP/s, respectively, which results in 1% to 11% improvements over the FlashAttention2 kernel for these sequence lengths. Nevertheless, for other sequence lengths, the FlashAttention2 CUDA kernel outperformed our implementation on average by 3.79%. Lastly, we observed a tile quantization of the GPU at these sequence lengths, as the throughput dropped for all the kernels from the $N = 512$ to $N = 577$.

Furthermore, our static mixed-precision attention implementation consistently outperformed the reference Triton FP16 and PyTorch’s memory-efficient attention. Unfortunately, due to the dynamic quantization overhead, our dynamic mixed-precision kernel did not improve the performance over the reference Triton FP16 implementation. Nevertheless, for $N = 4096$, we observed a peak throughput equal to 160.7 TOP/s, almost twice the performance of the memory-efficient attention kernel. At the sequence lengths 197, 577, and 785 with the Triton FP16 implementation, we achieved 71.4, 123.4, and 128.5 TOP/s, respectively. This translates to improvements of 19.66%, 4.41%, and 2.41% with our static mixed-precision attention kernel. On average, our mixed-precision kernel outperformed the reference Triton FP16 implementation by 6.83%. In addition, we observed an average throughput improvement of 49.268% over all sequence lengths compared to PyTorch’s memory-efficient kernel.

5. EFFICIENT GPU KERNELS FOR MIXED-PRECISION VISION TRANSFORMERS

With sixteen heads ($h = 16$) in attention, we observed the highest throughput performance of our mixed-precision kernel at $N = 197$; see Figure 5.13. We observed speedup improvements of 75.33%, 55.69%, and 56.53% over PyTorch, Triton FP16, and FlashAttention2, respectively. In this benchmark, our static mixed-precision attention kernel attained better performance the FlashAttention kernel for $N = 785$, which improved throughput by 3.53%. For $N = 577$, FlashAttention achieved 0.46% higher throughput than our static mixed-precision kernel. On average, FlashAttention2 offered a 3.45% increase in speed compared to our mixed-precision attention kernel for the rest of the sequence lengths.

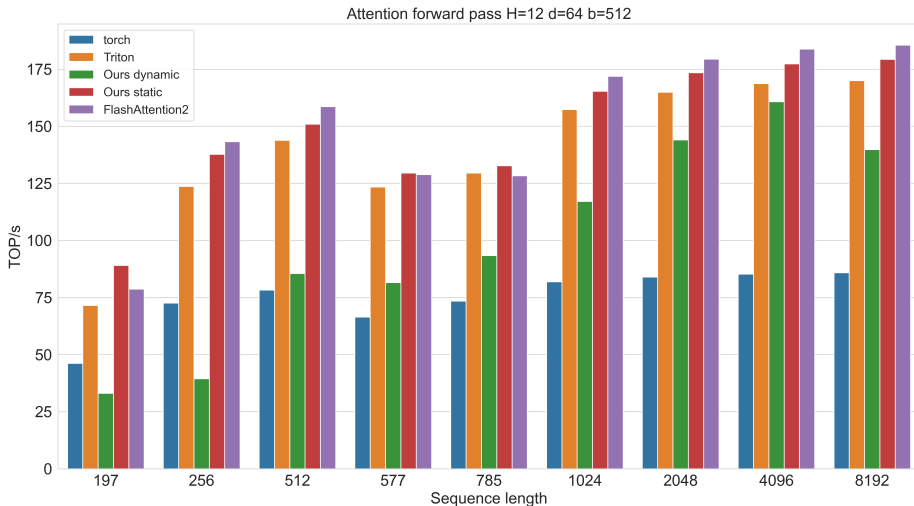


Figure 5.12: Performance comparison of various attention kernels with $H = 12$, $d = 64$, and $b = 512$: PyTorch scaled dot product attention, FP16 Triton reference implementation, dynamic mixed-precision attention, static quantization mixed-precision attention, and FlashAttention2.

Consistently, we found that PyTorch’s memory-efficient attention was the slowest across all the kernels. Similarly, we found that the dynamic mixed-precision attention kernel did not improve over the reference Triton FP16 implementation. The peak performance of dynamic quantized mixed-precision kernel was again observed for $N = 4096$ with throughput equal to 133.4 TOP/s. On average, our static quantized mixed-precision attention improved by 52.19% over PyTorch’s memory-efficient attention. Moreover, with a larger head size, we improved the average speedup over the Triton FP16 implementation to 10.64%.

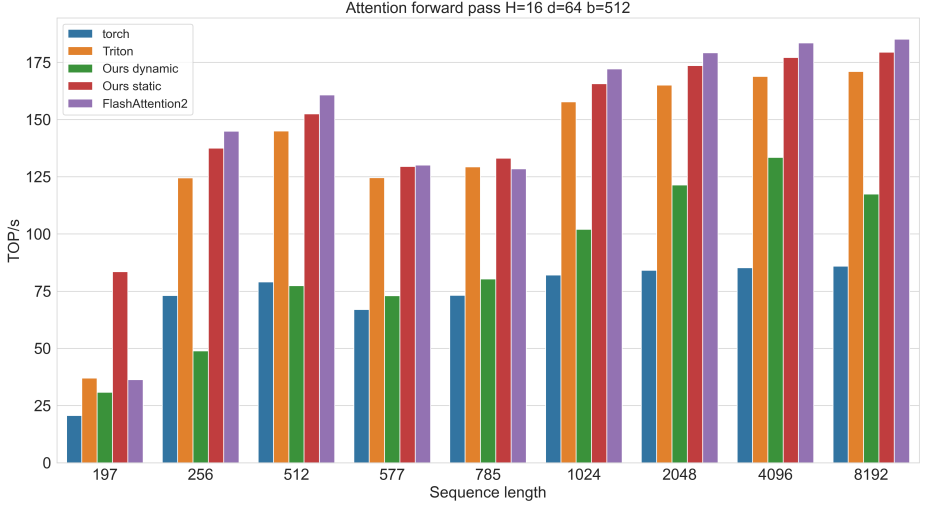


Figure 5.13: Performance comparison of various attention kernels with $H = 16$, $d = 64$, and $b = 512$: PyTorch scaled dot product attention, FP16 Triton reference implementation, dynamic mixed-precision attention, static quantization mixed-precision attention, and FlashAttention2.

5.8.3 Vision Transformers

In addition to the performance of the individual kernels, we evaluated the end-to-end numerical stability and throughput performance on the ViT models in the ImageNet1K classification task. Table 5.2 presents the top-1 accuracy obtained by running the calibration steps over five runs on the ImageNet1K. We note, consistent with our findings in previous chapters, that the highest quantization error was observed in the ViT-S model with patch size $P = \{16, 32\}$ during static quantization. Nevertheless, we observed that adding the static quantized mixed-precision attention did not incur additional quantization error. Moreover, in the case of the ViT-S/16/384 and ViT-S/32/384, the top-1 accuracy slightly improved. In the case of dynamic quantization, as expected, the quantization error was less pronounced than in static quantization.

Unfortunately, the dynamic quantized mixed-precision attention incurred additional quantization errors for the ViT-S. Table 5.3 presents the throughput speedup of static quantization compared to FP32 on the ImageNet1K dataset. For ViT-S/16, we observed a speedup of 3.4x and 3.55x for input images of dimensions

224×224 and 384×384 , respectively. Meanwhile, for ViT-S/32, we noted 3.32x and 2.9x improvement over the FP32 for 224×224 and 384×384 input image sizes, respectively. We note that the overhead of quantization and dequantization might impact the performance of the end-to-end measurements.

With ViT-B models, we observed a less pronounced quantization error in the static quantized kernels, except for ViT-B/32/224. Moreover, in the case of ViT-B/16/384, adding the static quantized mixed-precision attention kernel stabilized the quantization error. Similarly, we found that the dynamic quantization (INT8-D) achieved better top-1 accuracy than the static quantization (INT8). Furthermore, the addition of the dynamically quantized mixed-precision attention kernel did not have an impact as strong on the top-1 accuracy as in the case of the ViT-S models. Within the ViT-B models, we observed a speedup of at least 5.08x over FP32 with ViT-B/32/224 up to 5.91x with ViT-B/16/384; see Table 5.3.

The largest throughput improvements and the smallest quantization error were observed in the ViT-L/16/224 and ViT-L/32/384. we noted less than 1% quantization error in the best statically quantized models, both with and without statically quantized mixed-precision attention kernel. Slight improvements over static quantization were observed with dynamic quantization. Likewise, as in ViT-B, we noted a slight decrease in top-1 accuracy in the case of the added dynamically quantized mixed-precision attention kernel. Finally, with our statically quantized ViT-L/16/224 and ViT-L/32/384, we observed 7.34x and 6.94x higher throughput than the FP32.

Table 5.2: Comparison of top-1 accuracy on ImageNet1K validation dataset comparison between baselines and models with our kernels. We evaluated static (INT8) and dynamic (INT8-D) quantization of linear layers. INT8+A and INT8-D+A represent the addition of quantized mixed precision attention. Our runs are averaged over five repetitions. The results of PTQ4ViT and Zhang et al. are as reported by the authors. In bold, we mark the best static quantization result.

Model	FP32	INT8	INT8+A	INT8-D	INT8-D+A	PTQ4ViT	Zhang et al.
ViT-S/16/224	81.38	70.96 ± 0.48	69.60 ± 0.92	78.94 ± 0.08	76.68 ± 0.02	81.00	-
ViT-S/16/384	83.80	74.53 ± 0.47	75.46 $\pm$ 1.79	81.51 ± 0.00	80.42 ± 0.00	-	-
ViT-S/32/224	75.99	61.40 ± 0.64	60.33 ± 1.85	73.85 ± 0.03	71.12 ± 0.07	75.58	-
ViT-S/32/384	80.48	65.65 ± 0.86	67.07 $\pm$ 0.98	78.49 ± 0.01	74.08 ± 0.00	-	-
ViT-B/16/224	85.10	82.01 ± 0.53	82.09 ± 0.31	83.50 ± 0.00	83.34 ± 0.02	84.25	81.81
ViT-B/16/384	85.99	79.42 $\pm$ 6.59	82.53 ± 0.62	84.10 ± 0.02	84.02 ± 0.00	85.82	-
ViT-B/32/224	80.73	70.21 $\pm$ 1.79	69.47 ± 1.32	77.78 ± 0.00	76.66 ± 0.00	-	-
ViT-B/32/384	83.35	81.17 $\pm$ 0.15	80.83 ± 0.12	82.33 ± 0.00	80.33 ± 0.00	-	-
ViT-L/16/224	85.84	84.26 ± 0.21	84.13 $\pm$ 0.47	85.12 ± 0.01	85.08 ± 0.03	-	84.84
ViT-L/32/384	81.10	81.00 ± 0.00	80.97 $\pm$ 0.12	81.38 ± 0.00	81.03 ± 0.00	-	-

Table 5.3: Throughout speedup (FP32 vs ours static) of ViT models measured over the ImageNet1K validation dataset.

Model	Batch Size	Speedup (x)
ViT-S/16/224	2048	3.4
ViT-S/16/384	1024	3.55
ViT-S/32/224	2048	3.32
ViT-S/32/384	1024	2.9
ViT-B/16/224	2048	5.88
ViT-B/16/384	1024	5.91
ViT-B/32/224	2048	5.08
ViT-B/32/384	1024	5.22
ViT-L/16/224	2048	7.34
ViT-L/32/384	1024	6.94

5.8.4 Segment Anything Model

SAM is a general-purpose segmentation model that is able to segment the object in the image based on the prompts (points or bounding boxes). It consists of the heavy ViT-based image encoder and two lightweight networks - prompt encoder and mask decoder. We focused on the image encoder since this is where most of the computation in the model takes place. We used a QAttn torch.compile workflow to apply our dynamic and static quantized kernels for the image encoder.

Figure 5.14 shows the prediction performance (mIOU [%]) relative to number of images processed per second by the SAM. We have labeled all model sizes, starting with base (SAM-B), large (SAM-L), and huge (SAM-H). The blue line represents the baseline reference implementation in FP32. It reached 11.46, 4.62, and 2.66 images processed per second at 53.64%, 56.18%, and 58.09% mIOU, respectively.

With Segment Anything Fast per-channel dynamic quantization (orange line), we reproduced the improvement of the throughput to 32.59, 11.95, and 6.87 for SAM-B, SAM-L, and SAM-H, respectively. Moreover, it did not incur additional quantization errors for the models, which achieved 53.6%, 56.62%, and 58.21%, respectively.

Our per-channel dynamic quantized kernels implemented in Triton (INT8-D) achieved similar predictive performance to the baseline FP32 and INT8-D*, i.e., 53.77%, 56.62%, and 58.21% mIOU for SAM-B, SAM-L, and SAM-H, respectively. Moreover, we achieved the best throughput across all the benchmarks with our INT8-D for SAM-B and SAM-L, corresponding to 58.99 and 27.47 images per second, respectively. This is over 5.15 and 5.95 times more images processed per

second than the FP32 reference implementation. Compared to Segment Anything Fast, our implementation processed 1.81 and 2.30 times more images per second, respectively. Finally, we measured a throughput of 16.09 images per second with SAM-H, which is 6.05 and 2.34 times more images per second than the FP32 and Segment Anything Fast implementations, respectively.

Lastly, our per-channel static quantized matrix multiplication kernels resulted in the best mIOU at 53.91% and 57.08% for SAM-B and SAM-L, respectively, with 47.78 and 25.92 images processed per second. As a result, our implementation achieved 4.17 times and 5.61 times more images processed than the FP32 reference. In addition, we measured a throughput of 17.31 images per second with SAM-H, which translates to 6.50 times more images processed per second compared to the reference implementation. Unfortunately, the predictive performance dropped to 55.83% mIOU after static quantization. Nevertheless, our statically quantized SAM-H achieved higher throughput and better predictive power than the reference FP32 SAM-B.

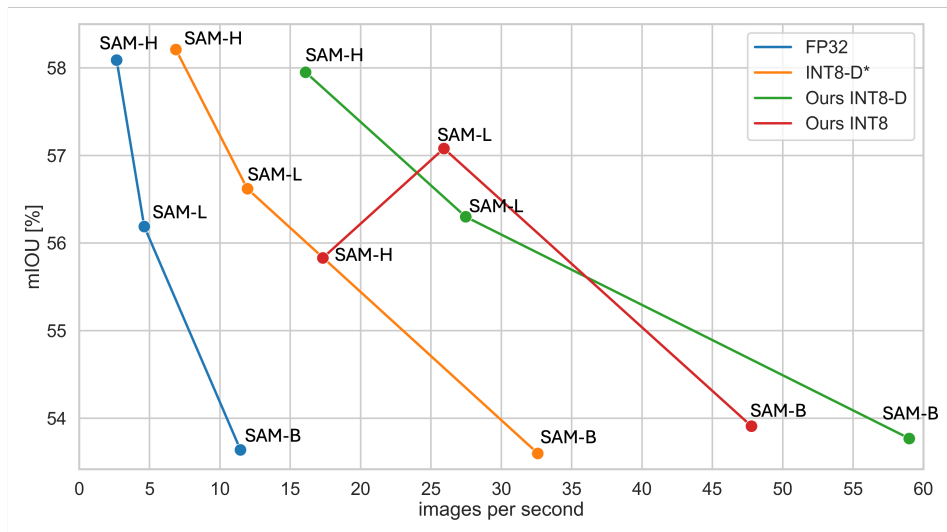


Figure 5.14: Relationship between predictive performance and image processing speed (images per second) for each SAM model, based on data type. Larger models show improved predictive power at the expense of reduced throughput. The blue line represents the reference FP32 SAM implementation, the orange line represents the Segment Anything Fast INT8-D*, while green and red lines are our dynamically and statically quantized kernels.

5.9 Summary and Conclusions

In this chapter, we have explored the frameworks and languages that could accelerate quantized ViTs on the GPU. We have identified Triton - a DSL with an MLIR compiler that was designed to accelerate the DL workloads on GPUs. Based on the Triton and PyTorch quantization ecosystem, we developed QAttn. This open-source framework implements a static and dynamic quantized matrix multiplication kernel and our proposed mixed-precision attention kernels. We presented the integration of this framework with torch.fx and the recent torch.compile quantization workflows.

We implemented a benchmark suite to test our kernel implementations. First, we evaluated the matrix multiplication kernel with dimensions representing various fully connected layers across the ViT model family. We compared these against the PyTorch FP32, the custom TVM INT8 kernel, and TensorRT INT8. While our static and dynamic quantized kernels underperformed for batch size $b = 1$, at larger batch sizes, they outperformed the closed-source custom TensorRT kernels.

Our proposed static mixed-precision kernel showed promising results, improving the throughput by 6.83% and 10.64% on average over the Triton FP16 implementation. In addition, our attention kernel exceeded or matched the performance of the FlashAttention2 CUDA kernel for the sequence lengths used in the ViT architecture.

We evaluated the QAttn framework on the ViT architecture with the ImageNet1K validation dataset. We evaluated scalar quantization for matrix multiplication with or without mixed-precision attention. We showed that we can achieve up to 7.34x speedup over the reference FP32 implementation with less than 1% quantization error. Moreover, we showed that the mixed-precision attention did not incur additional quantization error in most of the evaluated ViTs.

Finally, we extended the experiments to the instance segmentation with SAM. We compared our per-channel matrix multiplication kernel applied with the QAttn torch.compile workflow with baseline FP32 and dynamically quantized Segment Anything Fast implementations. We showed significant speedups (over five times more images processed per second) compared to both implementations without accuracy degradation at dynamic quantization. Unfortunately, our static per-channel quantization incurred a 3% quantization error to the SAM-H model. Nevertheless, our quantized SAM-H was 6.5x faster than the reference FP32 SAM-H. However, after applying static per-channel quantized kernels, SAM-B and SAM-L slightly improved their predictive performance alongside image processing speed - 4.17x and 5.61x faster than the reference FP32 implementation.

Chapter 6

Conclusions

In this chapter, we summarize and review the findings of the thesis. In addition, we propose an outlook on future work on the post-training quantization of ViT models. Due to the extensive amount of data and DL model sizes, the quantization became an exciting research direction. Moreover, it is evolving due to new model architectures like ViT or hardware support for new data types like FP8 or INT4. In this thesis, we tackled the post-training quantization of ViT models holistically. The main objective of the thesis was to design algorithms and methods that could leverage both the new architecture features and hardware acceleration. Starting from the comprehensive effect of quantization on novel ViT architectures to algorithms that allow the mitigation of the devastating impact of quantization on these architectures. Lastly, we proposed a novel type of attention-mixed-precision attention alongside an open-source framework to let researchers and practitioners work on quantizing ViTs.

6.1 Summary of Results

Challenges in Post-Training Quantization of Vision Transformers

In the first part of Chapter 3, we showed that the naive post-training quantization has a negative effect on the predictive power of the ViT and DeiT3 architectures. We noted that the DeiT and Swin Transformer were more robust to the quantization

than the previously mentioned architectures. Based on this insight and differences in the training procedures and model architecture, we hypothesized that the regularization applied during training might positively impact the quantization’s robustness. Next, we showed a correlation between the SQNR and the predictive power of the model after quantization. In this chapter, we have tackled the first research question of the thesis, where we proposed a method to compress the model up to 90% of the nodes with minimal loss of accuracy.

Hybrid Quantization

In Chapter 4, we proposed and evaluated a novel quantization algorithm. Based on the SQNR metric, we designed hybrid quantization algorithms that select either static or dynamic quantization for the linear layer in the ViT architecture. Our method improved the top-1 accuracy of 12/12 ViT, 3/6 DeiT, 6/6 DeiT3, and 6/6 Swin Transformer models compared to static quantization. In addition, we demonstrated reduced latency compared to dynamic quantization on an Intel Xeon 5218 Gold CPU, an Apple A15 Bionic mobile CPU, and an NVIDIA A100 GPU. Our method provides a new tool for a tradeoff between latency and predictive power of ViT models, fulfilling our second research objective.

Efficient GPU Kernels for Mixed-Precision Vision Transformers

Finally, in Chapter 5, we focused on bridging the gap between research and practical quantized workflows by making QAttn framework open-source, realizing the third research objective. This tool allows researchers and practitioners to evaluate their quantization methods on GPU accelerators. We showed that the kernels implemented in Triton DSL can match or outperform the closed-source TensorRT equivalent kernel. Our last research question was attained by design and implementation of the mixed-precision attention that closed the performance gap between the DSL kernel and the tuned CUDA kernel. Finally, we showed the speedups achieved by using our library for image classification and instance segmentation tasks. We achieved up to a 7.34 speedup over FP32 for ViT-L/16/224 and over 5x more images processed per second with the SAM model.

6.2 Outlook

In our thesis, we tackled the post-training quantization of ViTs using INT8. First, in the future, we need to address the optimal training procedure so that the produced DL model is robust enough for quantization. This could include exploring new data types such as FP8, FP6, and FP4. Second, modern accelerators also

support other data types, such as INT4. To further compress ViT architectures, a promising approach would be mixed-precision integer quantization, where the model is quantized to INT8 or INT4. However, this is challenging because the search space increases with the number of bit widths considered. Nevertheless, Triton DSL does not support INT4 natively, we could investigate weight and activation packing to mitigate this. Lastly, we should implement activations, normalization, and other operations to perform in a quantized regime to avoid dequantization.

Acronyms

A

ACIQ	Analytical Clipping for Integer Quantization
AWQ	Activation-Aware Weight Quantization
AxC	Approximate Computing

B

BFloat16	Brain Float 16-bit
----------	--------------------

C

CNN	Convolutional Neural Network
ConvNet	Convolutional Neural Network
COCO	Common Objects in Context dataset
CSC	Compressed sparse column
CPU	Central processing unit
CSR	Compressed sparse row
CUDA	Compute Unified Device Architecture
cuDNN	CUDA Deep Nerual Network
CV	Computer Vision

D

ACRONYMS

DeiT	Data efficient Image Transformer
DeiT3	Data efficient Image Transformer 3
DL	Deep Learning
DNN	Deep Neural Network
DSL	Domain Specific Language

F

FN	False Negative
FP	False Positive
FP32	Floating point 32-bit
FP16	Floating point 16-bit
FP8	Floating point 8-bit

G

GB	Gigabyte
GPU	Graphics Processing Unit
GELU	Gaussian Error Linear Unit
GEMM	General matrix multiplication
GPUSQ-ViT	GPU Friendly Sparisty and Quantization - Vision Transformers

H

HQ	Hybrid Quantization
----	---------------------

I

IEEE	Institute of Electrical and Electronics Engineers
INT32	signed 32-bit integer
INT4	signed 4-bit integer
INT8	signed 8-bit integer
IoU	Intersection over Union
IO	Input Output
IoT	Internet of Things

L

L2	Level 2 cache
LLM	Large Language Model
LayerNorm	Layer Normalization
LNorm	Layer Normalization
LSQ	Learned Step Size Quantization

M

mAP	mean Average Precision
MAC	Multiply-add operation
MHA	Multi Head Attention
mIOU	mean Intersection over Union
ML	Machine Learning
MLIR	Multi-Level Intermediate Representation Overview
MLP	Multi Layer Perceptron

N

NLP	Natural Language Processing
-----	-----------------------------

O

OPs	Operations count
-----	------------------

P

PTQ	Post-training Quantization
PTQ4ViT	Post-training Quantization for Vision Transformers
PQ	Panoptic Quality

Q

QAT	Quantization-aware training
-----	-----------------------------

R

ResNet	[Residual Network]
RELU	[Recitfied Linear Unit]

ACRONYMS

RGB red, green, and blue

S

SQNR Signal-to-quantization-noise ratio
SRAM static random-access memory
STE Straight-through estimation
Swin Shifted window

T

TFLOP/s Tera-floating point operations per second
TOP/s Tera operations per second
TP True Positive
TPU Tensor Processing Unit

U

UINT8 unsigned 8-bit integer

V

ViT Vision Transformer

X

XLA Accelerated Linear Algebra
XNOR Exclusive or

Bibliography

- [1] N. Maslej, L. Fattorini, R. Perrault, et al. *Artificial Intelligence Index Report 2024*. 2024. arXiv: 2405.19522 [cs.AI] (cited on p. 1).
- [2] M. Deitke, C. Clark, S. Lee, et al. *Molmo and PixMo: Open Weights and Open Data for State-of-the-Art Multimodal Models*. 2024. arXiv: 2409.17146 [cs.CV] (cited on p. 1).
- [3] Z. Chen, J. Wu, W. Wang, et al. “InternVL: Scaling up Vision Foundation Models and Aligning for Generic Visual-Linguistic Tasks”. In: *arXiv preprint arXiv:2312.14238* (2023) (cited on p. 1).
- [4] Z. Chen, W. Wang, H. Tian, et al. “How Far Are We to GPT-4V? Closing the Gap to Commercial Multimodal Models with Open-Source Suites”. In: *arXiv preprint arXiv:2404.16821* (2024) (cited on p. 1).
- [5] H. Touvron, T. Lavril, G. Izacard, et al. “Llama: Open and efficient foundation language models”. In: *arXiv preprint arXiv:2302.13971* (2023) (cited on pp. 1, 13, 23, 26).
- [6] H. Touvron, L. Martin, K. Stone, et al. “Llama 2: Open foundation and fine-tuned chat models”. In: *arXiv preprint arXiv:2307.09288* (2023) (cited on pp. 1, 13, 26).

- [7] A. Dubey, A. Jauhri, A. Pandey, et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI] (cited on p. 1).
- [8] A. Dosovitskiy, L. Beyer, A. Kolesnikov, et al. “An image is worth 16x16 words: Transformers for image recognition at scale”. In: *arXiv preprint arXiv:2010.11929* (2020) (cited on pp. 1, 13–15, 18, 34, 89).
- [9] L. Huang, W. Yu, W. Ma, et al. “A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions”. In: *arXiv preprint arXiv:2311.05232* (2023) (cited on p. 1).
- [10] A. Ometov and J. Nurmi. “Towards Approximate Computing for Achieving Energy vs. Accuracy Trade-offs”. In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2022, pp. 632–635. DOI: 10.23919/DATE54114.2022.9774538 (cited on pp. 1, 33).
- [11] T. Le Scao, A. Fan, C. Akiki, et al. “Bloom: A 176b-parameter open-access multilingual language model”. In: (2023) (cited on pp. 1, 13, 23, 26).
- [12] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort. “A white paper on neural network quantization”. In: *arXiv preprint arXiv:2106.08295* (2021) (cited on pp. 2, 17, 20, 23, 25, 38).
- [13] W. Dally. “High-performance hardware for machine learning”. In: *Nips Tutorial 2* (2015), p. 3 (cited on pp. 2, 21, 77).
- [14] Y. LeCun, Y. Bengio, and G. Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444 (cited on pp. 5, 12).
- [15] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016 (cited on pp. 5, 8, 12, 13).
- [16] T. M. Mitchell. *The discipline of machine learning*. Vol. 9. Carnegie Mellon University, School of Computer Science, Machine Learning ..., 2006 (cited on p. 5).
- [17] S. Ioffe and C. Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. pmlr. 2015, pp. 448–456 (cited on pp. 6, 12).

-
- [18] J. L. Ba, J. R. Kiros, and G. E. Hinton. “Layer normalization”. In: *arXiv preprint arXiv:1607.06450* (2016) (cited on pp. 6, 12, 14, 57).
 - [19] D. Hendrycks and K. Gimpel. “Gaussian error linear units (gelus)”. In: *arXiv preprint arXiv:1606.08415* (2016) (cited on pp. 6, 14, 57).
 - [20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536 (cited on p. 7).
 - [21] R. Balestrieri, M. Ibrahim, V. Sobal, et al. “A cookbook of self-supervised learning”. In: *arXiv preprint arXiv:2304.12210* (2023) (cited on p. 7).
 - [22] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255 (cited on pp. 8, 13, 18, 34, 58, 76, 89).
 - [23] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. “Microsoft coco: Common objects in context”. In: *Computer Vision—ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6–12, 2014, Proceedings, Part V 13*. Springer. 2014, pp. 740–755 (cited on pp. 10, 76, 90).
 - [24] A. Kirillov, K. He, R. Girshick, C. Rother, and P. Dollár. “Panoptic segmentation”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 9404–9413 (cited on p. 11).
 - [25] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (cited on pp. 12, 13).
 - [26] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012) (cited on pp. 12, 13, 26).
 - [27] S. Woo, S. Debnath, R. Hu, X. Chen, Z. Liu, I. S. Kweon, and S. Xie. “Convnext v2: Co-designing and scaling convnets with masked autoencoders”.

- In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2023, pp. 16133–16142 (cited on pp. 13, 18).
- [28] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. “Ssd: Single shot multibox detector”. In: *Computer Vision—ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*. Springer. 2016, pp. 21–37 (cited on p. 13).
- [29] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. “You only look once: Unified, real-time object detection”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788 (cited on p. 13).
- [30] K. He, G. Gkioxari, P. Dollár, and R. Girshick. “Mask r-cnn”. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2961–2969 (cited on p. 13).
- [31] D. P. Kingma and M. Welling. “Auto-encoding variational bayes”. In: *arXiv preprint arXiv:1312.6114* (2013) (cited on p. 13).
- [32] A. Brock, J. Donahue, and K. Simonyan. “Large scale GAN training for high fidelity natural image synthesis”. In: *arXiv preprint arXiv:1809.11096* (2018) (cited on p. 13).
- [33] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018) (cited on p. 13).
- [34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017) (cited on p. 13).
- [35] K. He, X. Zhang, S. Ren, and J. Sun. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (cited on pp. 14, 18, 26, 56).
- [36] 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*. PMLR. 2021, pp. 10347–10357 (cited on pp. 15, 16, 18, 34).
- [37] X. Zhai, A. Kolesnikov, N. Houlsby, and L. Beyer. “Scaling vision transformers”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022, pp. 12104–12113 (cited on pp. 15, 17, 23, 34).
- [38] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré. “FlashAttention: Fast and memory-efficient exact attention with io-awareness”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 16344–16359 (cited on pp. 16, 34).
- [39] T. Dao. “FlashAttention-2: Faster attention with better parallelism and work partitioning”. In: *arXiv preprint arXiv:2307.08691* (2023) (cited on pp. 16, 83, 89).
- [40] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo. “Swin transformer: Hierarchical vision transformer using shifted windows”. In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 10012–10022 (cited on pp. 16, 18, 34).
- [41] H. Touvron, M. Cord, and H. Jégou. “Deit iii: Revenge of the vit”. In: *European conference on computer vision*. Springer. 2022, pp. 516–533 (cited on pp. 17, 18, 34).
- [42] H. Touvron, M. Cord, A. Sablayrolles, G. Synnaeve, and H. Jégou. “Going deeper with image transformers”. In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 32–42 (cited on pp. 17, 34).
- [43] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229 (cited on pp. 17–19).
- [44] R. Krishnamoorthi. “Quantizing deep convolutional networks for efficient inference: A whitepaper”. In: *arXiv preprint arXiv:1806.08342* (2018) (cited on pp. 17, 25).

- [45] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell. “Bfloat16 Processing for Neural Networks”. In: *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. 2019, pp. 88–91. DOI: 10.1109/ARITH.2019.00022 (cited on pp. 18, 19).
- [46] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky. “NVIDIA A100 Tensor Core GPU: Performance and Innovation”. In: *IEEE Micro* 41.2 (2021), pp. 29–35. DOI: 10.1109/MM.2021.3061394 (cited on pp. 18, 19, 24, 29, 30, 77).
- [47] P. Micikevicius, D. Stosic, N. Burgess, et al. “Fp8 formats for deep learning”. In: *arXiv preprint arXiv:2209.05433* (2022) (cited on pp. 18, 19).
- [48] M. van Baalen, A. Kuzmin, S. S. Nair, et al. “Fp8 versus int8 for efficient deep learning inference”. In: *arXiv preprint arXiv:2303.17951* (2023) (cited on pp. 18, 19).
- [49] Y. Bengio, N. Léonard, and A. Courville. “Estimating or propagating gradients through stochastic neurons for conditional computation”. In: *arXiv preprint arXiv:1308.3432* (2013) (cited on pp. 20, 26).
- [50] M. Nagel, M. Fournarakis, Y. Bondarenko, and T. Blankevoort. “Overcoming oscillations in quantization-aware training”. In: *International Conference on Machine Learning*. PMLR. 2022, pp. 16318–16330 (cited on p. 20).
- [51] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. “Quantization and training of neural networks for efficient integer-arithmetic-only inference”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713 (cited on pp. 22, 25).
- [52] R. Bommasani, D. A. Hudson, E. Adeli, et al. “On the opportunities and risks of foundation models”. In: *arXiv preprint arXiv:2108.07258* (2021) (cited on p. 23).
- [53] E. Almazrouei, H. Alobeidli, A. Alshamsi, et al. “The falcon series of open language models”. In: *arXiv preprint arXiv:2311.16867* (2023) (cited on p. 23).

-
- [54] S. Zhang, S. Roller, N. Goyal, et al. “Opt: Open pre-trained transformer language models”. In: *arXiv preprint arXiv:2205.01068* (2022) (cited on pp. 23, 26).
 - [55] M. Dehghani, J. Djolonga, B. Mustafa, et al. “Scaling vision transformers to 22 billion parameters”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 7480–7512 (cited on p. 23).
 - [56] C. Saharia, W. Chan, S. Saxena, et al. “Photorealistic text-to-image diffusion models with deep language understanding”. In: *Advances in neural information processing systems* 35 (2022), pp. 36479–36494 (cited on p. 23).
 - [57] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever. “Zero-shot text-to-image generation”. In: *International conference on machine learning*. Pmlr. 2021, pp. 8821–8831 (cited on p. 23).
 - [58] D. Podell, Z. English, K. Lacey, A. Blattmann, T. Dockhorn, J. Müller, J. Penna, and R. Rombach. “Sdxl: Improving latent diffusion models for high-resolution image synthesis”. In: *arXiv preprint arXiv:2307.01952* (2023) (cited on p. 23).
 - [59] Y. Gao, Y. Xiong, X. Gao, et al. “Retrieval-augmented generation for large language models: A survey”. In: *arXiv preprint arXiv:2312.10997* (2023) (cited on p. 23).
 - [60] A. Kirillov, E. Mintun, N. Ravi, et al. “Segment anything”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2023, pp. 4015–4026 (cited on pp. 23, 76, 90).
 - [61] J. Gou, B. Yu, S. J. Maybank, and D. Tao. “Knowledge distillation: A survey”. In: *International Journal of Computer Vision* 129.6 (2021), pp. 1789–1819 (cited on p. 23).
 - [62] S. Masoudnia and R. Ebrahimpour. “Mixture of experts: a literature survey”. In: *Artificial Intelligence Review* 42 (2014), pp. 275–293 (cited on pp. 23, 24).
 - [63] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste. “Sparsity in deep learning: Pruning and growth for efficient inference and training in

- neural networks”. In: *Journal of Machine Learning Research* 22:241 (2021), pp. 1–124 (cited on pp. 23, 24).
- [64] G. Hinton, O. Vinyals, and J. Dean. “Distilling the knowledge in a neural network”. In: *arXiv preprint arXiv:1503.02531* (2015) (cited on p. 23).
- [65] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio. “FitNets: Hints for Thin Deep Nets”. In: *CoRR* abs/1412.6550 (2014) (cited on p. 23).
- [66] S. Zagoruyko and N. Komodakis. “Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer”. In: *arXiv preprint arXiv:1612.03928* (2016) (cited on p. 23).
- [67] M. Ranzinger, G. Heinrich, J. Kautz, and P. Molchanov. “AM-RADIO: Agglomerative Vision Foundation Model Reduce All Domains Into One”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024, pp. 12490–12500 (cited on p. 23).
- [68] A. Radford, J. W. Kim, C. Hallacy, et al. “Learning transferable visual models from natural language supervision”. In: *International conference on machine learning*. PMLR. 2021, pp. 8748–8763 (cited on p. 23).
- [69] M. Oquab, T. Darcet, T. Moutakanni, et al. “Dinov2: Learning robust visual features without supervision”. In: *arXiv preprint arXiv:2304.07193* (2023) (cited on p. 23).
- [70] T. G. Dietterich et al. “Ensemble learning”. In: *The handbook of brain theory and neural networks* 2.1 (2002), pp. 110–125 (cited on p. 24).
- [71] A. Q. Jiang, A. Sablayrolles, A. Roux, et al. “Mixtral of experts”. In: *arXiv preprint arXiv:2401.04088* (2024) (cited on p. 24).
- [72] A. Kuzmin, M. Nagel, M. Van Baalen, A. Behboodi, and T. Blankevoort. “Pruning vs quantization: which is better?” In: *Advances in neural information processing systems* 36 (2024) (cited on pp. 24, 25).
- [73] S. Yun and A. Wong. “Do all mobilenets quantize poorly? gaining insights into the effect of quantization on depthwise separable convolutional networks

- through the eyes of multi-scale distributional dynamics”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 2447–2456 (cited on p. 25).
- [74] M. Nagel, M. v. Baalen, T. Blankevoort, and M. Welling. “Data-free quantization through weight equalization and bias correction”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 1325–1334 (cited on p. 25).
- [75] R. Banner, Y. Nahshan, E. Hoffer, and D. Soudry. “ACIQ: ANALYTICAL CLIPPING FOR INTEGER QUAN”. In: *arXiv preprint arXiv:1810.05723* (2018) (cited on p. 25).
- [76] M. Nagel, R. A. Amjad, M. Van Baalen, C. Louizos, and T. Blankevoort. “Up or down? adaptive rounding for post-training quantization”. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 7197–7206 (cited on p. 25).
- [77] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. “Xnor-net: Imagenet classification using binary convolutional neural networks”. In: *European conference on computer vision*. Springer. 2016, pp. 525–542 (cited on p. 26).
- [78] A. Bulat and G. Tzimiropoulos. “Xnor-net++: Improved binary neural networks”. In: *arXiv preprint arXiv:1909.13863* (2019) (cited on p. 26).
- [79] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han, et al. “McuNet: Tiny deep learning on iot devices”. In: *Advances in neural information processing systems* 33 (2020), pp. 11711–11722 (cited on p. 26).
- [80] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han. “MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning”. In: *Annual Conference on Neural Information Processing Systems (NeurIPS)*. 2021 (cited on p. 26).
- [81] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han. “On-Device Training Under 256KB Memory”. In: (2022) (cited on p. 26).

- [82] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha. “Learned step size quantization”. In: *arXiv preprint arXiv:1902.08153* (2019) (cited on p. 26).
- [83] Y. Bhalgat, J. Lee, M. Nagel, T. Blankevoort, and N. Kwak. “Lsq+: Improving low-bit quantization through learnable offsets and better initialization”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*. 2020, pp. 696–697 (cited on p. 26).
- [84] P. Ramachandran, B. Zoph, and Q. V. Le. “Searching for activation functions”. In: *arXiv preprint arXiv:1710.05941* (2017) (cited on p. 26).
- [85] D. Misra. “Mish: A self regularized non-monotonic activation function”. In: *arXiv preprint arXiv:1908.08681* (2019) (cited on p. 26).
- [86] A. Polino, R. Pascanu, and D. Alistarh. “Model compression via distillation and quantization”. In: *arXiv preprint arXiv:1802.05668* (2018) (cited on p. 26).
- [87] G. Team, R. Anil, S. Borgeaud, et al. “Gemini: a family of highly capable multimodal models”. In: *arXiv preprint arXiv:2312.11805* (2023) (cited on p. 26).
- [88] M. Reid, N. Savinov, D. Teplyashin, et al. “Gemini 1.5: Unlocking multi-modal understanding across millions of tokens of context”. In: *arXiv preprint arXiv:2403.05530* (2024) (cited on p. 26).
- [89] B. Adler, N. Agarwal, A. Aithal, et al. “Nemotron-4 340B Technical Report”. In: *arXiv preprint arXiv:2406.11704* (2024) (cited on p. 26).
- [90] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. “Qlora: Efficient finetuning of quantized llms”. In: *Advances in Neural Information Processing Systems* 36 (2024) (cited on p. 26).
- [91] E. Frantar, S. Ashkboos, T. Hoefer, and D. Alistarh. “Gptq: Accurate post-training quantization for generative pre-trained transformers”. In: *arXiv preprint arXiv:2210.17323* (2022) (cited on p. 27).

-
- [92] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer. “GPT3.int8(): 8-bit Matrix Multiplication for Transformers at Scale”. In: *Advances in Neural Information Processing Systems*. Ed. by A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho. 2022 (cited on p. 27).
 - [93] T. Dettmers and L. Zettlemoyer. “The case for 4-bit precision: k-bit inference scaling laws”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 7750–7774 (cited on pp. 27, 77).
 - [94] J. Lin, J. Tang, H. Tang, et al. “AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration”. In: *Proceedings of Machine Learning and Systems* 6 (2024), pp. 87–100 (cited on p. 27).
 - [95] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer. “Llm. int8 (): 8-bit matrix multiplication for transformers at scale”. In: *arXiv preprint arXiv:2208.07339* (2022) (cited on pp. 27, 77).
 - [96] Y. Lin, T. Zhang, P. Sun, Z. Li, and S. Zhou. “FQ-ViT: Post-Training Quantization for Fully Quantized Vision Transformer”. In: *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*. 2022, pp. 1173–1179 (cited on pp. 28, 37, 59).
 - [97] Z. Liu, Y. Wang, K. Han, W. Zhang, S. Ma, and W. Gao. “Post-training quantization for vision transformer”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 28092–28103 (cited on p. 28).
 - [98] Z. Yuan, C. Xue, Y. Chen, Q. Wu, and G. Sun. “PTQ4ViT: Post-training quantization framework for vision transformers with twin uniform quantization”. In: *arXiv preprint arXiv:2111.12293* (2021) (cited on pp. 28, 37, 90).
 - [99] Z. Wang, C. Wang, X. Xu, J. Zhou, and J. Lu. “Quantformer: Learning Extremely Low-Precision Vision Transformers”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.7 (2023), pp. 8813–8826. DOI: 10.1109/TPAMI.2022.3229313 (cited on p. 28).
 - [100] Z. Li and Q. Gu. “I-ViT: Integer-only Quantization for Efficient Vision Transformer Inference”. In: *Proceedings of the IEEE/CVF International*

- Conference on Computer Vision (ICCV)*. 2023, pp. 17065–17075 (cited on p. 29).
- [101] Z. Zhang, B. He, and Z. Zhang. “Practical Edge Kernels for Integer-Only Vision Transformers Under Post-training Quantization”. In: *Proceedings of Machine Learning and Systems* 5 (2023) (cited on pp. 29, 90).
- [102] J. Moon, D. Kim, J. Cheon, and B. Ham. “Instance-Aware Group Quantization for Vision Transformers”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2024, pp. 16132–16141 (cited on p. 29).
- [103] C. Yu, T. Chen, Z. Gan, and J. Fan. “Boost vision transformer with gpu-friendly sparsity and quantization”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2023, pp. 22658–22668 (cited on p. 29).
- [104] A. C. Elster and T. A. Haugdahl. “Nvidia Hopper GPU and Grace CPU Highlights”. In: *Computing in Science & Engineering* 24.2 (2022), pp. 95–100. DOI: 10.1109/MCSE.2022.3163817 (cited on pp. 29, 30).
- [105] *NVIDIA Blackwell Architecture Technical Overview*. NVIDIA. URL: <https://resources.nvidia.com/en-us-blackwell-architecture> (visited on 07/01/2024) (cited on pp. 29, 30).
- [106] *Deploying Transformers on the Apple Neural Engine*. Apple Machine Learning Research. URL: <https://machinelearning.apple.com/research/neural-engine-transformers> (cited on p. 29).
- [107] N. Jouppi, G. Kurian, S. Li, et al. “TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings”. In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 2023, pp. 1–14 (cited on p. 29).
- [108] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner. “AI Accelerator Survey and Trends”. In: *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 2021, pp. 1–9. DOI: 10.1109/HPEC49654.2021.9622867 (cited on p. 29).

-
- [109] J. Nickolls, I. Buck, M. Garland, and K. Skadron. “Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?” In: *Queue* 6.2 (2008), pp. 40–53 (cited on p. 30).
 - [110] C. Lattner and V. Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International symposium on code generation and optimization, 2004. CGO 2004*. IEEE. 2004, pp. 75–86 (cited on p. 30).
 - [111] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. “cuDNN: Efficient primitives for deep learning”. In: *arXiv preprint arXiv:1410.0759* (2014) (cited on p. 30).
 - [112] M. Martineau, P. Atkinson, and S. McIntosh-Smith. “Benchmarking the NVIDIA V100 GPU and Tensor Cores”. In: *Euro-Par 2018: Parallel Processing Workshops*. Ed. by G. Mencagli, D. B. Heras, V. Cardellini, et al. Cham: Springer International Publishing, 2019, pp. 444–455. ISBN: 978-3-030-10549-5 (cited on p. 30).
 - [113] H. Vanholder. “Efficient inference with TensorRT”. In: *GPU Technology Conference*. Vol. 1. 2. 2016 (cited on p. 30).
 - [114] NVIDIA. *TensorRT*. <https://github.com/NVIDIA/TensorRT> (cited on p. 30).
 - [115] S. Migacz. “8-bit inference with TensorRT”. In: *GPU technology conference*. Vol. 2. 4. 2017, p. 5 (cited on pp. 30, 89).
 - [116] NVIDIA. *TensorRT-LLM*. <https://github.com/NVIDIA/TensorRT-LLM> (cited on pp. 30, 77).
 - [117] NVIDIA. *FasterTransformer*. <https://github.com/NVIDIA/FasterTransformer> (cited on pp. 30, 77).
 - [118] *Torch: a scientific computing framework for LUA JIT*. <https://github.com/torch/torch7> (cited on p. 30).

- [119] T. T. D. Team, R. Al-Rfou, G. Alain, et al. “Theano: A Python framework for fast computation of mathematical expressions”. In: *arXiv preprint arXiv:1605.02688* (2016) (cited on p. 30).
- [120] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *arXiv preprint arXiv:1408.5093* (2014) (cited on p. 30).
- [121] M. Abadi, A. Agarwal, P. Barham, et al. “Tensorflow: Large-scale machine learning on heterogeneous distributed systems”. In: *arXiv preprint arXiv:1603.04467* (2016) (cited on p. 30).
- [122] A. Paszke, S. Gross, F. Massa, et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019) (cited on pp. 30, 59, 89).
- [123] J. Bradbury, R. Frostig, P. Hawkins, et al. *JAX: composable transformations of Python+NumPy programs*. <http://github.com/google/jax>. Version 0.3.13. 2018 (cited on p. 30).
- [124] A. Sabne. *XLA : Compiling Machine Learning for Peak Performance*. 2020 (cited on p. 30).
- [125] G. Van Rossum and F. L. Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995 (cited on p. 30).
- [126] G. Gerganov. *ggml*. <https://github.com/ggerganov/ggml> (cited on p. 30).
- [127] W. Kwon, Z. Li, S. Zhuang, et al. “Efficient Memory Management for Large Language Model Serving with PagedAttention”. In: *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. 2023 (cited on p. 30).
- [128] T. Chen, T. Moreau, Z. Jiang, et al. “{TVM}: An automated {End-to-End} optimizing compiler for deep learning”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 578–594 (cited on pp. 30, 89).

- [129] A. Hannun, J. Digani, A. Katharopoulos, and R. Collobert. *MLX: Efficient and flexible machine learning on Apple silicon*. Version 0.0. 2023 (cited on p. 31).
- [130] C. Lattner, M. Amini, U. Bondhugula, et al. “MLIR: Scaling compiler infrastructure for domain specific computation”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pp. 2–14 (cited on p. 31).
- [131] P. Tillet, H.-T. Kung, and D. Cox. “Triton: an intermediate language and compiler for tiled neural network computations”. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2019, pp. 10–19 (cited on pp. 31, 59, 78, 89).
- [132] Modular. *Mojo: The Mojo Programming Language*. 2023 (cited on p. 31).
- [133] C. Sun, A. Shrivastava, S. Singh, and A. Gupta. “Revisiting unreasonable effectiveness of data in deep learning era”. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 843–852 (cited on p. 34).
- [134] T. Ridnik, E. Ben-Baruch, A. Noy, and L. Zelnik-Manor. “Imagenet-21k pretraining for the masses”. In: *arXiv preprint arXiv:2104.10972* (2021) (cited on p. 34).
- [135] A. Steiner, A. Kolesnikov, X. Zhai, R. Wightman, J. Uszkoreit, and L. Beyer. “How to train your vit? data, augmentation, and regularization in vision transformers”. In: *arXiv preprint arXiv:2106.10270* (2021) (cited on p. 34).
- [136] T. Darcet, M. Oquab, J. Mairal, and P. Bojanowski. “Vision transformers need registers”. In: *arXiv preprint arXiv:2309.16588* (2023) (cited on p. 36).
- [137] Y. Bondarenko, M. Nagel, and T. Blankevoort. “Quantizable Transformers: Removing Outliers by Helping Attention Heads Do Nothing”. In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine. Vol. 36. Curran Associates, Inc., 2023, pp. 75067–75096 (cited on p. 36).

- [138] R. Wightman. *PyTorch Image Models*. <https://github.com/rwightman/pytorch-image-models>. 2019. DOI: 10.5281/zenodo.4414861 (cited on pp. 36, 58).
- [139] D. Cai, Q. Wang, Y. Liu, Y. Liu, S. Wang, and M. Xu. “Towards ubiquitous learning: A first measurement of on-device training performance”. In: *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*. 2021, pp. 31–36 (cited on p. 55).
- [140] A. Das, Y. D. Kwon, J. Chauhan, and C. Mascolo. “Enabling on-device smartphone gpu based training: Lessons learned”. In: *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. IEEE. 2022, pp. 533–538 (cited on p. 55).
- [141] S. Li, C. Tian, K. Tam, R. Ma, and L. Li. “Breaking On-device Training Memory Wall: A Systematic Survey”. In: *arXiv preprint arXiv:2306.10388* (2023) (cited on p. 55).
- [142] N. Nazari and M. E. Salehi. “Inter-Layer Hybrid Quantization Scheme for Hardware Friendly Implementation of Embedded Deep Neural Networks”. In: *Proceedings of the Great Lakes Symposium on VLSI 2023*. 2023, pp. 193–196 (cited on p. 56).
- [143] P. Panda. “Quanos: adversarial noise sensitivity driven hybrid quantization of neural networks”. In: *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. 2020, pp. 187–192 (cited on p. 56).
- [144] N. P. Pandey, M. Nagel, M. van Baalen, Y. Huang, C. Patel, and T. Blankevoort. “A Practical Mixed Precision Algorithm for Post-Training Quantization”. In: *arXiv preprint arXiv:2302.05397* (2023) (cited on p. 56).
- [145] D. Khudia, J. Huang, P. Basu, S. Deng, H. Liu, J. Park, and M. Smelyanskiy. “FBGEMM: Enabling High-Performance Low-Precision Deep Learning Inference”. In: *arXiv preprint arXiv:2101.05615* (2021) (cited on pp. 59, 78).

-
- [146] M. Dukhan, Y. Wu, and H. Lu. *QNNPACK: Open source library for optimized mobile deep learning*. Engineering at Meta. URL: <https://engineering.fb.com/2018/10/29/ml-applications/qnnpack/> (cited on p. 59).
 - [147] J. Reed, Z. DeVito, H. He, A. Ussery, and J. Ansel. “Torch. fx: Practical program capture and transformation for deep learning in python”. In: *Proceedings of Machine Learning and Systems* 4 (2022), pp. 638–651 (cited on p. 76).
 - [148] J. Ansel, E. Yang, H. He, et al. “PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation”. In: (2024) (cited on pp. 76, 89).
 - [149] M. van Baalen, A. Kuzmin, S. S. Nair, et al. “FP8 versus INT8 for efficient deep learning inference”. In: *arXiv preprint arXiv:2303.17951* (2023) (cited on p. 77).
 - [150] V. Thakkar, P. Ramani, C. Cecka, et al. *CUTLASS*. Version 3.0.0. Jan. 2023 (cited on p. 77).
 - [151] T. Dettmers, M. Lewis, S. Shleifer, and L. Zettlemoyer. “8-bit Optimizers via Block-wise Quantization”. In: *9th International Conference on Learning Representations, ICLR* (2022) (cited on p. 77).
 - [152] NVIDIA. *Torch-TensorRT*. <https://github.com/pytorch/TensorRT> (cited on p. 89).

