QAttn: Efficient GPU Kernels for mixed-precision Vision Transformers

Piotr Kluska
IBM Research Europe
Universitat Politècnica de València
klu@zurich.ibm.com

Adrián Castelló
Universitat Politècnica de València
adcastel@disca.upv.es

Florian Scheidegger
IBM Research Europe
eid@zurich.ibm.com

A. Cristiano I. Malossi
IBM Research Europe
acm@zurich.ibm.com

Enrique S. Quintana-Ortí
Universitat Politècnica de València
quintana@disca.upv.es

Abstract

Vision Transformers have demonstrated outstanding performance in Computer Vision tasks. Nevertheless, this superior performance for large models comes at the expense of increasing memory usage for storing the parameters and intermediate activations. To accelerate model inference, in this work we develop and evaluate integer and mixed-precision kernels in Triton for the efficient execution of two fundamental building blocks of transformers –linear layer and attention– on graphics processing units (GPUs). On an NVIDIA A100 GPU, our kernel implementations of Vision Transformers achieve a throughput speedup of up to 7x compared with reference kernels in PyTorch floating-point single precision (FP32). Additionally, the accuracy for the ViT Large model top-1 drops by less than one percent on the ImageNet1K classification task. We also observe up to 6x increased throughput by applying our kernels to the Segment Anything Model image encoder while keeping the mIOU close to the FP32 reference on the COCO2017 dataset for static and dynamic quantization. Furthermore, our kernels demonstrate improved speed to the TensorRT INT8 linear layer, and we improve the throughput of base FP16 (half precision) Triton attention on average by up to $19 \pm 4.01\%$. *We have open-sourced the QAtnn framework, which is tightly integrated with the PyTorch quantization workflow* https://github.com/IBM/qattn.

1. Introduction

Recent advancements in Foundation Models (FM) [3], both in Natural Language Processing (NLP) [38, 42, 45] and Computer Vision (CV) [17, 27, 40], have extended the predictive performance of deep learning models. Nevertheless, these advances come at a cost in terms of computational requirements and memory resources. Currently, the baseline reference for FM is a transformer architecture enhanced with an attention mechanism [48]. Initially designed for NLP, transformers have been adapted for CV, resulting in the development of Vision Transformers (ViT) [17]. ViTs are encoder-only models that are typically self-supervised, pre-trained on a large amount of data, and later adapted for downstream tasks such as image classification, object detection, or instance segmentation. Similar to large language models [45], ViTs come in different sizes, depending on the number of layers and, as a result, parameters, which vary from millions to 22 billion [12]. As a result, the largest model requires a dedicated accelerator with sufficient memory to process the data. The large size of ViTs makes them appropriate candidates for compression methods such as quantization, but outliers in intermediate activations pose a challenge [4, 11]. Quantization, a compression technique that reduces the number of bits, converts computation and data from "continuous" (floating point) to discrete (integer). Integer 8-bit (INT8) inference is faster and more energy-efficient than its floating-point counterparts, but the limited range in which we can represent values makes it susceptible to quantization errors during computation that may affect the final accuracy of the deep learning model [20, 30].

Modern hardware includes dedicated units that support efficient matrix multiplication in 8-bit and 16-bit floating point (FP8 and {FP16; BFloat16}, resp.) as well as INT8 [6, 28, 46]. To take full advantage of these computing units, users must use a specialized hardware runtime application programming interface (API) such as CUDA [31], which requires significant programming skills to take full advantage of the graphics proccessing unit's (GPU) capabilities. Most deep learning frameworks, such as PyTorch [39], implement operations that can be accelerated with GPUs. However, because it is a general-purpose framework, there may be missing implementations that are used, for example, in quantization.

Table 1. Operation count (OPs) by percentage for ViT models and latency measurements. We count only nodes used during the inference. Attention is counted as one operation even though it contains two matrix multiplications with softmax. Number of layers, OPs, and latency are provided as a percentage of the ViT large model. We provide latency numbers 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

Currently, there are specialized frameworks for accelerating selected models on dedicated platforms, such as ggml [18] or MLX [19]. ggml supports multiple hardware targets (x86, ARM, CUDA, Metal, etc.) and selected models, but requires the hardware API to accelerate inference. MLX is a dedicated tensor framework for Apple's silicon chips to accelerate inference of deep learning models using the chip's unified memory. However, while it has a familiar Python API, extending MLX with new kernels requires C++ or Metal API knowledge. Alternatively, compilers such as Apache TVM [7] allow custom kernels to be written and optimised using general techniques; and next compiled plus fine-tuned for selected target hardware, such as a GPUs, ARM multicore processors, etc. Nevertheless, writing developer-guided kernels in TVM is still a challenging chore that requires a significant understanding of the target platform and, especially, the TVM compiler stack.

In contrast to the previous approaches, Triton is a domain-specific language and compiler designed for developing efficient CUDA kernels in Python [44]. Triton compiles Python routines into device code with LLVM using MLIR [23, 24]. Moreover, it provides an auto-tuning option to hint the compiler about the sizes of the tensor operands. Lastly, it directly integrates with PyTorch [39] and Jax [5] (via Pallas), making it an ideal candidate for efficiently and quickly developing new GPU kernels as it hides the complexity of the CUDA language.

In our work, we focus on the quantization of the most relevant layers and operations of ViT –the linear layer (matrix multiplication) and the attention module. These are the most dominant operation nodes in this architecture, accounting for over 99% of the floating-point operations per second (FLOPS) in the ViT large and taking up to 96.34% of the time to process the data depending on the batch size, as shown in Table 1. In doing so, we make the following specific contributions:

- We develop QAttn (Quantized attention, pronounced like katana), a Python framework with efficient GPU implementations of the linear layer and attention. Our integration with the quantization workflow in torch.fx [41] and torch.compile [2] allows in-place replacement of PyTorch modules. (At this point, we note that the current version of PyTorch does not support lowering the quantized model to the GPU.)
- We evaluate the performance of matrix multiplication with PyTorch, TVM, TensorRT [47], and our proposed method over different sizes of linear layers present in ViT models. 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 $19 \pm 4.01\%$ on average.
- We verify the numerical stability of our quantized and mixed-precision kernels on the ImageNet1K [13] validation dataset across different ViT models.The experiments with ViT-Large models show less than 1% degradation in top-1 accuracy, with 75% weight reduction and up to 7x speedup compared to FP32.
- We extend our matrix multiplication to support variations of quantization granularity, *e.g.*, scalar and per-channel.
- We apply dynamic and static quantized kernels to the image encoder of the Segment Anything Model [22]. We achieve over 5x more images processed per second for the base and large image encoders without mIOU drop over the COCO2017 [25] validation set for static and dynamic quantization. For the huge model, we achieve over 6x more images per second and no accuracy drop for dynamic quantization, while for static, the mIOU dropped only by 2.66%

The rest of the paper is structured as follows. In Section 2 we review the relevant literature in this area. In Section 3, we introduce QAttn, starting with a brief description of the deep learning operations and quantization. We also present an example code that shows how the framework is integrated with PyTorch. In Section 4, we present our experimental setup and results, with subsections devoted to matrix multiplication, attention, ViT performance for mixed-precision inference, and SAM performance after quantization. We discuss our findings in Section 5, followed by a review of the limitations and future work in Section 6. Finally, we conclude our work in Section 7.

2. Related Work

The progression in hardware and software technology has led to more powerful FMs. Nevertheless, the challenge re-

mains of efficiently deploying FM by utilizing target hardware resources. Currently, the standard format for inference is FP32 or FP16. While efficient inference in FP16 is supported on modern hardware via specialized cores, e.g., Tensor Cores in NVIDIA GPUs, INT8 is supposed to achieve twice the theoretical throughput [33]. Moreover, INT8 consumes two times less memory for loading model weights and intermediate activations and requires significantly less energy [8, 46].

TensorRT is an inference framework developed by NVIDIA that can efficiently run deep learning models, in multiple data formats (FP32, FP16, and INT8), both on discrete GPUs and Jetson boards [29, 35]. TensorRT integrates with TensorFlow [1], PyTorch [39], and ONNX Runtime [37] via open-source libraries that aim to minimize the effort to port the models to the TensorRT runtime. However, TensorRT is a closed-source library.

Another framework developed by NVIDIA is FasterTransformer [32]. This is an open-source tool that contains specialized CUDA kernels for popular NLP and CV transformer models for FP and INT inference. However, NVIDIA recently deprecated FasterTransformer in favor of TensorRT-LLM, which supports only NLP models [34].

PyTorch 2.0 natively supports CPU-only quantized deep learning models via FBGEMM [21]. For PyTorch, GPU support is enabled by lowering the model to TensorRT runtime via the TorchTensorRT package [36]. While TorchTensorRT is an open-source package, it utilizes closed-source kernels from TensorRT. However, PyTorch has a flexible backend configuration that we can extend to integrate INT8 GPU inference.

bitsandbytes is an open-source framework for efficient training using mixed-precision data types [15]. The authors developed custom CUDA and Triton kernels for efficient INT8 and sub-byte types FP4 and normalized float 4-bit [14]. Those kernels enable efficient training of the models as the states of the optimizers as well as the weights are quantized [16]. The main target are large language models (LLMs). However, they also implemented the Switchback algorithm that could effectively train vision-language CLIP ViT-L using the INT8 linear layer with up to 25% speedup compared to BFloat16 [50].

Lastly, methods like FQ-ViT [26] or PTQ4ViT [51] tried to find optimal scaling factors for the weights of ViT to avoid performance degradation. However, those methods simulated INT8 quantization to run the methods efficiently on GPUs. Nevertheless, Zhang *et al*. [52] demonstrated the implementation of INT8 approximated Softmax, GELU, and Layer Normalization. They claim to achieve up to 5x speedups compared to FP32 on Apple Silicon A13 and M1 chips.

3. QAttn

In this section, we provide a detailed description of various aspects of the QAttn framework. Concretely, we start by briefly discussing the quantization process, followed by a definition of the quantized matrix multiplication and fused attention operations. Finally, we discuss the integration of the QAttn framework in the PyTorch quantization workflow.

3.1. Quantization

Quantization is an approximate method that can help to reduce memory and energy consumption for deep learning tasks. As part of this process, the data format is compressed by, for example, reducing the number of bits from the standard 32 in deep learning algorithms to 16 or even 8 bits. Additionally, we can switch from floating-point data and arithmetic to integer to further reduce memory and energy usage. However, by doing so, we introduce quantization errors due to the reduced bit width and the use of integer arithmetic. Fortunately, deep learning models are fair candidates for quantization since they are heavily overparameterized, which helps minimize the quantization error.

In our study, we consider symmetric INT8 post-training static and dynamic quantization. In this scheme, the quantization function

$$\mathcal{Q}(X) = \lfloor \frac{X}{s} \rceil, \tag{1}$$

scales the entries of the input tensor X element-wise, and then rounds each to the nearest integer. The scaling factor s is calculated, based on the maximum absolute value in the input tensor X, as

$$s = \frac{\max(\text{abs}(X))}{(q_{\max} - q_{\min})/2}, \tag{2}$$

where $q_{\min}$ and $q_{\max}$ respectively denote the lower bound and the upper bound of the data type. (For example, in the case of INT8 and two's complement representation, these respectively correspond to -128 and +127). The inverse operation to quantization is dequantization, which calculates the approximate value in the continuous format as

$$\mathcal{Q}'(\mathcal{Q}(X), s) = \mathcal{Q}(X) \cdot s. \tag{3}$$

Considering only symmetric quantization, we do not include zero point in (1). While we can calculate the scaling factor s for parametric layers, for intermediate activations we need to either statically estimate that parameter based on the calibration set or, alternatively, choose the value of s dynamically during inference. The higher flexibility of dynamic quantization introduces an overhead to calculate scaling factors based on the data range of the inputs. Moreover, we can consider different granularities for computing the scaling factor. Given a multi-dimensional tensor, we can

apply a scalar quantization scale, which is the most storage-efficient. In this case for the whole tensor, we store an additional four bytes for the FP32 scale. In comparison, depending on the model's sensitivity to quantization, we can compute per-channel scaling factors. For example. for an $m \times n$ matrix A, if we consider quantization per row, we add m FP32 values to dequantize the matrix.

3.2. Operations

3.2.1 Linear

The linear layer is the fundamental component in the ViT architecture. This type of module performs a matrix multiplication involving the learned weights W and the input activation:

$$C = \mathcal{C}(X, W^T) = X \cdot W^T, \tag{4}$$

where X is the input matrix, of size $m \times n$, and W is the weight matrix, of size $k \times n$. The individual entries of the $m \times k$ result matrix C in (4) are thus given by a linear combination between the entries in the rows of X and W^T:

$$c_{ij} = \sum_{p=1}^{n} x_{ip} w_{jp}. \tag{5}$$

In *static quantization*, we quantize both X and W but also need to re-quantize the output to INT8:

$$Q^{\mathrm{S}} = \mathcal{Q}(\,\mathcal{C}(\mathcal{Q}(X),\, \mathcal{Q}(W^T))\,) \tag{6}$$

so that, for the entries of the result,

$$q_{ij}^{\mathrm{S}} = s^{\,\mathrm{S}} \sum_{p=1}^{n} \mathcal{Q}(x_{ip})\, \mathcal{Q}(w_{jp}), \tag{7}$$

where $s^{\,\mathrm{S}}$ is a static re-quantization scaling factor that is computed as a linear combination of the scaling factors for X, W and C [20]:

$$s^{\,\mathrm{S}} = \frac{s_X\, s_W}{s_C}. \tag{8}$$

Note that, in order to avoid overflow during the INT8 matrix multiplication, we use an INT32 accumulator.

In *dynamic quantization*, the output is instead an FP16 or FP32 matrix:

$$C^{\,\mathrm{D}} = \mathcal{C}(\,\mathcal{Q}(X),\, \mathcal{Q}(W^T)\,), \tag{9}$$

with its output entries given by

$$c_{ij}^{\mathrm{D}} = s^{\mathrm{D}} \sum_{p=1}^{n} \mathcal{Q}(x_{ip})\, \mathcal{Q}(w_{jp}), \tag{10}$$

and the dynamic scaling factor

$$s^{\mathrm{D}} = s_X\, s_W. \tag{11}$$

3.2.2 Attention

The attention mechanism is an indispensable element of the transformer that performs a pivotal function in NLP and CV tasks. It operates on three input tensors, namely query (Q), key (K), and value (V), all of dimension $n \times d$, where n denotes the context length and d corresponds to the head size. The process recurs over h heads and a batch of size b.

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:

$$S = \frac{\mathcal{C}(Q, K^T)}{\sqrt{d}}. \tag{12}$$

The attention weights are then obtained by applying the softmax function to the result:

$$P = \mathrm{softmax}(S), \tag{13}$$

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

$$O = \mathcal{C}(P, V). \tag{14}$$

3.2.3 Mixed-precision Attention

Dao *et al*. introduced Flash Attention and Flash Attention-2 [9, 10] with kernels designed to accelerate the forward pass of the attention mechanism by exploiting the hardware of data-parallel GPUs. Starting from these kernels, in this work, we developed static and dynamic mixed-precision attention in the Triton language, described next.

Similar to matrix multiplication, we have to calculate scaling factors for input and output in static quantization. To ensure the numerical stability of attention, we conduct the initial matrix multiplication in INT8, and subsequently de-quantize it to FP32:

$$S = \frac{s_Q\, s_K}{\sqrt{d}} \mathcal{C}(\mathcal{Q}(Q), \mathcal{Q}(K^T)), \tag{15}$$

for calculating attention weights, using softmax in floating point arithmetic:

$$P = \mathrm{softmax}(S). \tag{16}$$

For the second matrix multiplication, we also de-quantize the values matrix V to calculate it in floating point arithmetic. For static mixed precision, the output is of INT8 data type,

$$O^{\,\mathrm{S}} = \mathcal{Q}(\mathcal{C}(P, \mathcal{Q}'(\mathcal{Q}(V), s_V))). \tag{17}$$

Meanwhile, for dynamic mixed precision, the output is in FP16,

$$O^{\,\mathrm{D}} = \mathcal{C}(P, \mathcal{Q}'(\mathcal{Q}(V), s_V)). \tag{18}$$

3.3. PyTorch Integration

QAttn is a kernel library that can efficiently deploy mixed-precision ViT models. It implements statically and dynamically quantized linear layers and mixed-precision attention modules. It is designed to work with the PyTorch FX quantization workflow (via added backend configuration) and automatically captures and lowers the graph to supported GPU kernels. QAttn supports conventional as well as new FP types, such as FP32, FP16, and BF16, and executes Triton kernels in INT8 or INT8/FP16. Once the model is converted, it behaves like a torch nn.Module and can be applied in the same places where the previous model was used. Moreover, QAttn supports the experimental torchdynamo export path and PyTorch 2.0 quantization workflow. The modules and functions are lowered to default PyTorch operations representation (ATen) and then replaced with supported kernels, while the rest is kept as native PyTorch operations. In Algorithm 1, we present a simple example of how to utilize QAttn for static quantization of the model.

Algorithm 1 Example usage of QAttn static quantization with the torch.compile quantization workflow.

```
import torch
from torch._export import capture_pre_autograd_graph
from torch.ao.quantization.quantize_pt2e import (
    prepare_pt2e,
    convert_pt2e,
)
import qattn
from qattn.pt2e.quantizer import (
    QAttnQuantizer,
    get_default_qattn_quantization_config,
)

# initialize quantizer
quantizer = QAttnQuantizer()
quantizer.set_module_type(
    torch.nn.Linear,
    get_default_qattn_quantization_config(
        per_channel=True,
        is_dynamic=False,
        input_per_channel=True,
    ),
)

# initialize model and sample
model = ...
sample = ...
# export and prepare the model
exported_model = capture_pre_autograd_graph(
    model,
    (sample,),
)
prepared_model = prepare_pt2e(
    exported_model.cpu(),
    quantizer,
).cuda()

# calibrate for static quantization
...
converted_model = convert_pt2e(
    prepared_model,
    fold_quantize=True,
)
# invoke lowering via torch compile
model = torch.compile(prepared_model, backend="qattn")
_ = model(sample)
```

4. Experimental Results

4.1. Experiment setup

We used PyTorch 2.2.1, Triton 2.2.0, TensorRT 8.6.1 and TVM 0.13.0. The measurements were performed on an NVIDIA A100 GPU with an 80-GB HBM2e RAM. We utilized the torch.fx package for capturing and quantizing the models' graphs. While the torch.fx module might not be able to capture the graph if there are control loops or conditional statements, it is effective for ViT graphs. For the Segment Anything Model (SAM) [22] image encoder, we utilized the PyTorch 2.0 dynamo graph capture mechanism. We measured operation performance using TFLOP/s (tera floating-point operations per second). We use the same number of arithmetic for all algorithms, corresponding to those performed by the FP32 version, even though the quantized version, of those arithmetic opeartions are performed in integer arithmetic. We run the operation 1000 times with 100 warm-up steps with L2 cache flushed in between measurements. For consistency throughout the following discussion, we will refer to both as TOP/s.

We identified distinct linear layers and attention shapes across various models and assessed the raw performance of these operations with varying batch size $b = \{1; 32; 64; 128; 256; 512; 1,024; 2,048\}$. Specifically, we compared the performance of PyTorch FP32 with TensorRT INT8, our implementations in TVM (static INT8), and Triton for matrix multiplication, with static and dynamic INT8 quantization.

We evaluated the models' performance using ImageNet1k validation dataset [13] after quantization. The metric reported is the top-1 accuracy over 1,000 labels averaged over 5 runs. Moreover, we reported the throughput speedup of our quantized and mixed-precision kernels compared to the native FP32 PyTorch kernel. We calibrated the models for static quantization using 2,000 random samples from the ImageNet1K training dataset. We experimented with ViT models of different parameter sizes (s), patch sizes (p), and input image sizes (i), denoted as ViT-$s/p/i$. Concretely, we consider three different ViT sizes: small (ViT-S), base (ViT-B), and large (ViT-L); patch sizes of 16, 32; and image size of 224, 384. Our evaluation tests are named, e.g., as ViT-L/16/224. The model weights and code were fetched from the *timm* package [49].

Similarly, for instance segmentation, we utilized the COCO2017 validation dataset [25]. We report the mean intersection over union (mIOU) metric. We dynamically and statically quantize linear layers in the SAM image encoder. We evaluate ViT-B, ViT-L, and ViT-H image encoders of SAM. For SAM quantization we apply per-channel quanti-

zation both for activations and weights.

4.2. Matrix Multiplication

Figure 1 presents the representative linear layers of ViT-B/16/224 and ViT-L/32/384. On an NVIDIA A100 GPU, we observed that for a batch size equal to 1, the highest TOP/s rate was achieved by the INT8 TensorRT. However, for batch sizes greater than 1, our dynamically quantized matrix multiplication kernel achieved better throughput than FP32 native PyTorch kernel. Furthermore, when comparing statically quantized linear kernels, those implemented in Triton outperformed their TensorRT counterparts. For ViT-B/16/224, our implementation achieved 308 TOP/s, while TensorRT attained 184 TOP/s. For a batch size greater than or equal to 256, our kernel delivered 460 TOP/s compared to 221 TOP/s for TensorRT. Similarly, for ViT-L/32/384 with a batch size equal to 2048, we achieved 481 TOP/s compared to 266 TOP/s for TensorRT. For ViT-B/16/224 with TensorRT INT8, we achieved 359 TOP/s over all sizes compared to 190 TOP/s TensorRT, while for ViT-L/32/384, we attained 224 TOP/s compared to 381 TOP/s on average for TensorRT and Triton, respectively. The implementation of matrix multiplication in TVM using the tensor expressions (te) application programming interface (API) delivered 17 TOP/s.

4.3. Attention

We evaluated the performance of our attention kernels without a causal mask against PyTorch's memory efficient attention FP16, Fused Attention Triton FP16, and Flash Attention 2 (v2.5.6). We fixed batch size to $b = 512$, head dimension $d = 64$, variable sequence length, and number of heads $h = 12$ or $h = 16$ present in ViT-B and ViT-L. Unlike [9, 10], we do not shrink the batch size as a function of sequence length.

The designers of the ViT architecture selected "odd numbers for the sequence length", *e.g.*, 197, 577, and 785. For those values, we observed a drop in the kernels' performance, except for our statically mixed-precision attention implementation when the sequence length is equal to 197. In that case, we achieved twice the performance compared to FlashAttention2. Moreover, we matched its performance for the 577 sequence length and for 785, we outperformed this kernel. We also improved the performance compared to the reference Triton FP16 implementation by an average of $19 \pm 4.02\%$ for 16 heads.

4.4. Vision Transformers

As part of the experiments, we tested the numerical stability of our quantized kernels on ViTs using the ImageNet1K validation dataset. In Table 2, we present throughput speedup over FP32 models with our static linear layer and mixed-precision attention. We observed speedups over 3x for ViT-S and up to 7x for ViT-L. Our findings showed that the models benefit from dynamic quantization for linear layers and mixed-precision attention, with most models experiencing minimal accuracy degradation; see Table 3. As expected, we observed a low variance of the dynamic quantization compared to the static variant due to the exact scaling factor calculated for each input. The quantization error accumulated due to rounding and, small ViT models are most vulnerable to static quantization. Nonetheless, with more complex quantization methods, such as PTQ4ViT, we could achieve accuracy on par with FP32. Additionally, models such as ViT-L had lost up to 1% of top-1 accuracy while reducing weight size by 75%.

Table 2. Throughout speedup (FP32 vs ours static) of ViT models measured over 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

4.5. SAM

We observed no performance degradation when applying static and dynamic quantization for SAM's base and large image encoders. The mIOU slightly increased while processing over 5.15x and 5.94x more images per second; see Tables 4 and 5. Our dynamic kernels achieved increased speedups by 1.8x and 2.3x compared to PyTorch operations for the base and large variants. Unfortunately, the static implementation has achieved lower throughput for base and large variants. However, it should be noted that the inputs and outputs of kernels had to be similarly quantized and dequantized as in the dynamic quantization scenario. Nevertheless, we achieved up to 6.50 more images per second processed for the huge model compared to the baseline floating point model.

5. Discussion

Developing efficient kernels for inference requires expert hardware-specific knowledge. Consequently, adopting a compiler stack designed for deep learning, such as TVM or Triton, is appealing because it lowers the threshold to extract high performance from the target hardware. Nev-

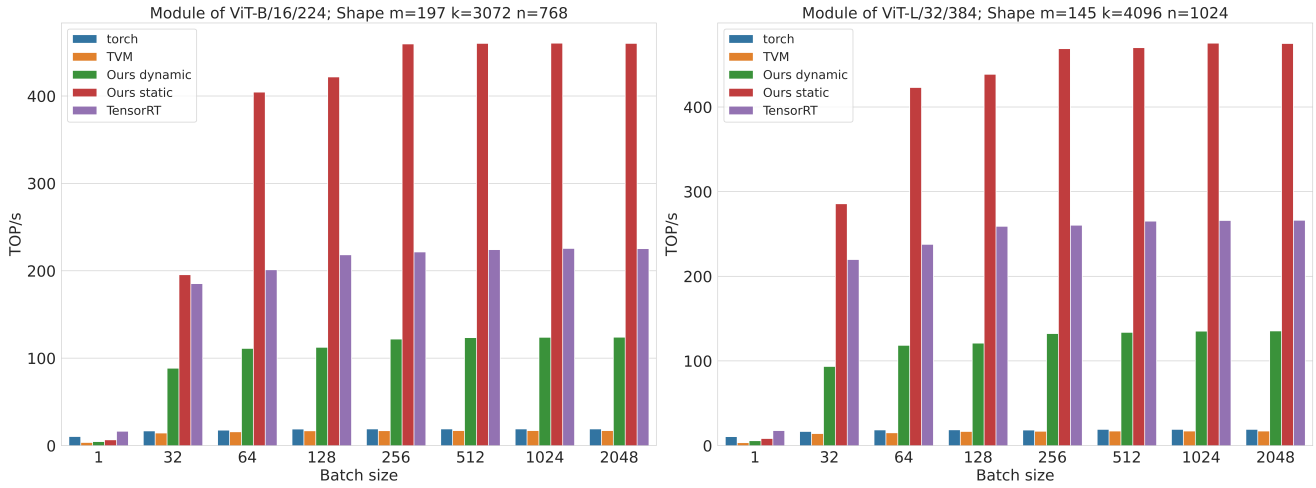

Figure 1. Matrix multiply with dimensions m, n, and k over various batch sizes. Reference Pytorch runs in FP32 and TensorRT in INT8. Our dynamic version accepts input FP16 with weights in INT8; our static implementation runs in INT8.

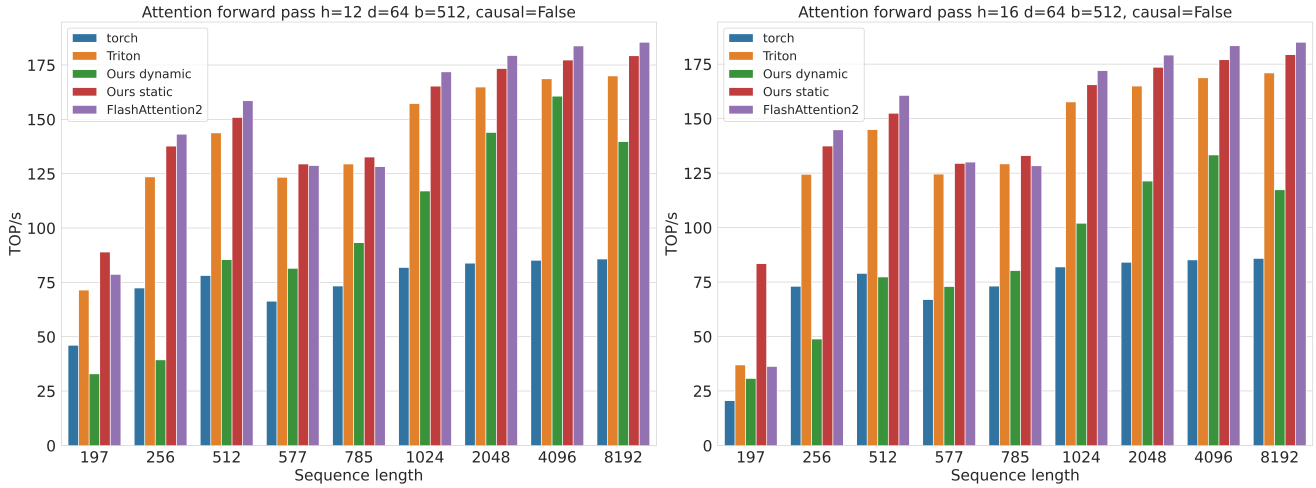

Figure 2. Performance of various attention kernels: torch scaled dot product attention, FP16 triton reference implementation, dynamic mixed-precision attention, static quantization mixed-precision attention, and FlashAttention2. The batch size was fixed to 512, the number of heads to 12 or 16, and the head dim to 64. The sequence length varies from 197 to 8k.

ertheless, for user adoption, the stack should have concise documentation, feature a simple API, and support specialized cores like Tensor Core. Unfortunately, we couldn't leverage this type of computational resource with the TVM API, which explains why the performance in our experiments with this framework flattens at 17 TOP/s.

The advantage of Triton is direct integration with deep learning frameworks that enables researchers and developers to iterate and integrate custom CUDA kernels into their codebase quickly. In the near future, Triton is expected to support more target hardware, such as AMD ROCm GPUs and Intel XPU, making the custom kernels portable. In the experiments, our kernels implemented in Triton outperformed TensorRT closed-source INT8 matrix multiplication achieving up to 481 TFLOP/s compared to 266 TFLOP/s for ViT-L/32/384 with batch size equal to 2048. Moreover, for the 16 heads with a batch size of 512, we could demonstrate that mixed-precision attention could improve the Triton reference implementation and match the performance of the custom FlashAttention2 CUDA kernel.

We demonstrated that the INT8 kernels implemented in Triton are a viable alternative to CPU PyTorch and TensorRT. Moreover, extending the PyTorch quantization workflow with QAttn allows researchers to develop new kernels and quantization algorithms accelerated by GPUs for ViTs.

Table 3. Top-1 accuracy on ImageNet1K validation dataset comparison between baselines and models with our kernels. We compare INT8 only linear with the option to add mixed-precision attention both in static and dynamic quantization (INT8-D) scenarios. 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. With ((+)), we also include mixed-precision attention.

Model	FP32	INT8	INT8$^+$	INT8-D	INT8-D$^+$	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 4. mIOU (%) performance on COCO2017 validation dataset. With ((*)), we denote the reproduced results of Segment Anything Fast dynamic quantization (without torch.compile) [43]. With INT8 and INT8-D, we represent only static and dynamic quantized linear layers.

Data type	SAM-B	SAM-L	SAM-H
FP32	53.64	56.18	58.09
INT8-D*	53.6	56.62	**58.21**
INT8-D	53.77	56.3	57.95
INT8	**53.91**	**57.08**	55.83

Table 5. Images per second processed by SAM models measured over COCO2017 validation dataset with batch size equal to 32. SAM with FP32 is the baseline implementation [22] with batch size equal to 1. We reproduced results of FP32 and INT8-D*.

Data Type	SAM-B	SAM-L	SAM-H
FP32	11.46	4.62	2.66
INT8-D*	32.59	11.95	6.87
INT8-D	**58.99**	**27.47**	16.09
INT8	47.78	25.92	**17.31**

We observed speedup of up to 7.34 in inference throughput versus FP32 for ViT-L and up to 6.5x more processed images per second for SAM-H, even after accounting for quantization and dequantization overhead. However, we believe that modifying the architecture of the ViT family to use tensor shapes that are integer multiples of 32 would improve performance even further.

6. Limitations and Future Work

This study only focuses on the ViT architecture as the base architecture of the vision transformers family. Nonetheless, the kernels described in this paper carry over to other transformer models, such as DeiT and Deit3. Introducing just two kernels adds an overhead of quantization and dequantization between activations and normalization layers. This issue needs to be addressed in future work. Furthermore, fusing QAttn with an inductor compiler to generate efficient kernels for general operations and CUDA graphs could improve the overall speed of torchdynamo-captured models.

7. Conclusions

We have shown that the implementation of INT8 kernels in Triton can be competitive with TensoRT. In addition, the mixed-precision attention can match the performance of the highly optimized CUDA FlashAttention2 implementation. We believe that our open-source inference framework directly integrated with PyTorch will enable researchers to innovate, prototype, and validate their ideas for efficient inference more quickly, for example, with sub-byte types such as INT4. In our setup, the top-1 performance of the ViT-B and ViT-L models was not significantly degraded by applying quantization to the INT8 linear layer and mixed-precision attention on the ImageNet1K classification task. In addition, we demonstrated that the per-channel quantization of the SAM image encoder did not affect the performance of the models, which achieved up to 6.5x and 2.5x times higher throughput, respectively, than the FP32 and INT8 PyTorch implementations.

Acknowledgments. The authors gratefully acknowledge 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, https://www.apropos-itn.eu/).

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016. 3

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. 2024. 2

[3] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021. 1

[4] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. Quantizable transformers: Removing outliers by helping attention heads do nothing. *arXiv preprint arXiv:2306.12929*, 2023. 1

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs. http://github.com/google/jax, 2018. 2

[6] Neil Burgess, Jelena Milanovic, Nigel Stephens, Konstantinos Monachopoulos, and David Mansell. Bfloat16 processing for neural networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 88–91. IEEE, 2019. 1

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, 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)*, pages 578–594, 2018. 2

[8] William Dally. High-performance hardware for machine learning. *Nips Tutorial*, 2:3, 2015. 3

[9] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023. 4, 6

[10] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022. 4, 6

[11] Timothée Darcet, Maxime Oquab, Julien Mairal, and Piotr Bojanowski. Vision transformers need registers. *arXiv preprint arXiv:2309.16588*, 2023. 1

[12] Mostafa Dehghani, Josip Djolonga, Basil Mustafa, Piotr Padlewski, Jonathan Heek, Justin Gilmer, Andreas Peter Steiner, Mathilde Caron, Robert Geirhos, Ibrahim Alabdulmohsin, et al. Scaling vision transformers to 22 billion parameters. In *International Conference on Machine Learning*, pages 7480–7512. PMLR, 2023. 1

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. 2, 5

[14] Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. In *International Conference on Machine Learning*, pages 7750–7774. PMLR, 2023. 3

[15] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022. 3

[16] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. *9th International Conference on Learning Representations, ICLR*, 2022. 3

[17] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 1

[18] Georgi Gerganov. ggml. https://github.com/ggerganov/ggml. 2

[19] Awni Hannun, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. MLX: Efficient and flexible machine learning on apple silicon. https://github.com/ml-explore, 2023. 2

[20] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry 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*, pages 2704–2713, 2018. 1, 4

[21] Daya Khudia, Jianyu Huang, Protonu Basu, Summer Deng, Haixin Liu, Jongsoo Park, and Mikhail Smelyanskiy. Fbgemm: Enabling high-performance low-precision deep learning inference. *arXiv preprint arXiv:2101.05615*, 2021. 3

[22] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C Berg, Wan-Yen Lo, et al. Segment anything. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4015–4026, 2023. 2, 5, 8

[23] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004. 2

[24] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021. 2

[25] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence 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, pages 740–755. Springer, 2014. 2, 5

[26] Yang Lin, Tianyu Zhang, Peiqin Sun, Zheng Li, and Shuchang Zhou. Fq-vit: Post-training quantization for fully quantized vision transformer. *arXiv preprint arXiv:2111.13824*, 2021. 3

[27] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 10012–10022, 2021. 1

[28] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, et al. FP8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022. 1

[29] Szymon Migacz. 8-bit inference with tensorrt. In *GPU technology conference*, page 5, 2017. 3

[30] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021. 1

[31] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008. 1

[32] NVIDIA. FasterTransformer. https://github.com/NVIDIA/FasterTransformer, . 3

[33] NVIDIA. NVIDIA A100 GPUs power the modern data center. https://www.nvidia.com/en-us/data-center/a100/, . 3

[34] NVIDIA. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM, . 3

[35] NVIDIA. TensorRT. https://github.com/NVIDIA/TensorRT, . 3

[36] NVIDIA. Torch-TensorRT. https://github.com/pytorch/TensorRT, . 3

[37] ONNX. ONNX Runtime. https://github.com/microsoft/onnxruntime, 2018. 3

[38] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023. 1

[39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019. 1, 2, 3

[40] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021. 1

[41] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. Torch. fx: Practical program capture and transformation for deep learning in python. *Proceedings of Machine Learning and Systems*, 4:638–651, 2022. 2

[42] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022. 1

[43] Pytorch Team. pytorch-labs/segment-anything-fast. https://github.com/pytorch-labs/segment-anything-fast, 2023. 8

[44] Philippe Tillet, Hsiang-Tsung Kung, and David 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*, pages 10–19, 2019. 2

[45] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023. 1

[46] Mart van Baalen, Andrey Kuzmin, Suparna S Nair, Yuwei Ren, Eric Mahurin, Chirag Patel, Sundar Subramanian, Sanghyuk Lee, Markus Nagel, Joseph Soriaga, et al. FP8 versus INT8 for efficient deep learning inference. *arXiv preprint arXiv:2303.17951*, 2023. 1, 3

[47] Han Vanholder. Efficient inference with tensorrt. In *GPU Technology Conference*, 2016. 2

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. 1

[49] Ross Wightman. Pytorch image models. https://github.com/rwightman/pytorch-image-models, 2019. 5

[50] Mitchell Wortsman, Tim Dettmers, Luke Zettlemoyer, Ari Morcos, Ali Farhadi, and Ludwig Schmidt. Stable and low-precision training for large-scale vision-language models. *arXiv preprint arXiv:2304.13013*, 2023. 3

[51] Zhihang Yuan, Chenhao Xue, Yiqi Chen, Qiang Wu, and Guangyu Sun. Ptq4vit: Post-training quantization for vision transformers with twin uniform quantization. In *European Conference on Computer Vision*, pages 191–207. Springer, 2022. 3

[52] Zining Zhang, Bingsheng He, and Zhenjie Zhang. Practical edge kernels for integer-only vision transformers under post-training quantization. *Proceedings of Machine Learning and Systems*, 5, 2023. 3