Optimization of Efficient Transformer for High Resolution Images

Ankit Rajvanshi
Computer Engineering
New York University
New York City, USA
ar7996

Gaurav Kuwar
Computer Engineering
New York University
New York City, USA
gk2657

***Abstract*—There exists plenty of state-of-the-art architectures which analyse images efficiently with great accuracy but all of them suffered from one common limitation that they are unable to analyse high resolution images accurately and efficiently. Hence, in [1], they introduced HCT for High resolution images and we tried to optimize it further by leveraging the capabilities of HPC in this work. Code repository can be found at https://github.com/sudoDollar/OptimizedHCT**

***Index Terms*—High Resolution, HPC, Distributed, Quantization, Training, Inference**

I. Introduction

A. Background and Motivation

Vision Models like ResNets and Vision Transformer (ViT) can indeed be attributed to several factors, one of which is their simplicity and ease of implementation and understanding. However, their limitations with high-resolution images have spurred the development of alternative architectures like [1] High-resolution Convolutional Transformer (HCT). ViT model struggles with high-resolution images because these images translate into a large number of tokens, i.e., a large sequence length. These large sequences are computationally expensive for the self-attention layers due to their quadratic complexity. On the other hand, a ResNet — or a pure CNN — struggles with high-resolution images because its effective receptive field is not large enough to cover a high-resolution image.

This new architecture, HCT, combines both the efficiency of convolutional layers and the global attention of transformers.

B. Problem Statement

While HCT is already efficient when compared with other vision models as it reduces quadratic complexity of self-attention to linear, it still needs to be optimized further because input size is very huge (4000x4000 pixels per image) and to make model viable in real time production systems.

C. Objectives and Scope

The primary objective of this work is to optimize the training and inference of HCT to enable faster and more effective processing of high-resolution images in applications such as medical imaging and satellite image analysis.

We will explore the techniques like distributed learning across multiple GPUs, Just-in-Time Compilation, quantization, deployment of models to non python environments. We will perform thorough evaluations tests to assess the performance, accuracy, and efficiency of the optimized HCT architecture particularly concerning its suitability for real-time applications and large-scale image analysis tasks.

II. Literature Review

A. High Resolution Convolutional Transformer

HCT in [1] adopts a ResNet-style architecture, depicted in fig 1, featuring a 7x7 convolutional stem followed by five stages. This design offers several advantages:

1) It enables flexibility in input resolutions, allowing HCT to seamlessly transition between different resolutions without requiring architectural modifications
2) The stages generate a feature pyramid with varying resolutions, accommodating both classification and detection tasks.

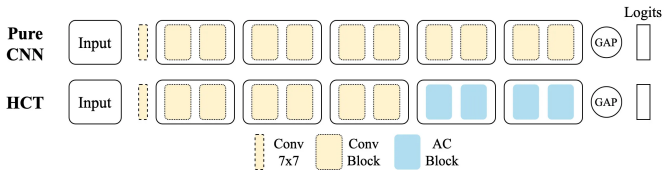

Fig. 1. A Pure CNN vs. HCT architecture. GAP denotes global average pooling

In a conventional CNN, each stage typically comprises pure convolutional blocks like BasicBlock and Bottleneck. However, HCT diverges from this approach by employing pure convolutional blocks only in the early stages. Instead, it integrates transformers in later stages. These transformers endow HCT with a global receptive field, thereby reducing the depth needed to process high-resolution images.

To seamlessly integrate an efficient transformer into a ResNet stage, HCT introduces an attention-convolutional (AC) block, illustrated in fig. 2. The AC block merges the global receptive field of transformers with the effectiveness of convolutional layers. Importantly, the AC block operates on spatial feature maps, facilitating its integration into various computer

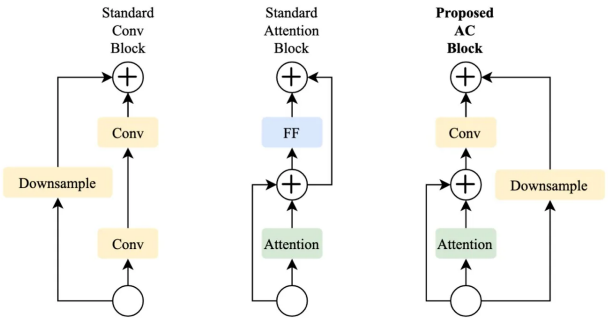

Fig. 2. Attention-Convolutional (AC) block with both spatial downsampling and long range attention capabilities

vision architectures. HCT harnesses Performers for the attention operation, which are efficient transformers featuring a linear-complexity self-attention layer.

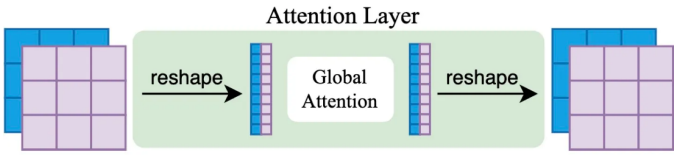

Fig. 3. Self-attention layer flattens the input feature maps before applying global self-attention, then reshapes the result into spatial feature maps

By employing a straightforward reshape operation, the self-attention layer is extended to all spatial locations within a feature map, as depicted in Fig. 3. This global attention mechanism offers several benefits: (1) It simplifies the configuration by reducing the number of hyperparameters, in contrast to block-wise methods such as local attention. (2) It operates without assuming a fixed number of tokens (feature maps' size), thereby accommodating varying input resolutions akin to CNNs, without additional complexities.

B. Large Minibatch SGD

Deep learning flourishes with large neural networks and extensive datasets. However, the extended training times associated with larger networks and datasets can hinder research and development progress. Distributed synchronous SGD offers a potential solution by distributing SGD minibatches across multiple parallel workers. However, to ensure efficiency, each worker's workload must be substantial, leading to an increase in the SGD minibatch size. In [2], they empirically demonstrate that using large minibatches on the ImageNet dataset poses optimization challenges. However, when these challenges were addressed, the trained networks show good generalization. Specifically, they observe no loss of accuracy when training with large minibatch sizes of up to 8192 images. To achieve this, they adopt a hyperparameter-free linear scaling rule i.e. *When the minibatch size is multiplied by k, multiply the learning rate by k* to adjust learning rates based on minibatch size and introduce a new warmup scheme to tackle optimization issues early in training.

III. Methodology

A. Data Collection and Preprocessing

In paper [1], they used original medical images to train the model but due to restrictions of data usage due to privacy issues, we were not able use original dataset and hence we will use non-medical [3] LIU4K (based on natural scenes), publicly available dataset, for our work. This dataset is small in size but we have no other option as there are only very few public datasets comprising High Resolutions Images.

LIU4K dataset contains high resolution 1600 training images and 260 validation images from 4 categories namely Animal, Building, Street and Mountain.

The images in dataset are of different resolutions and hence before feeding the images to the model we need to resize all the images to 4000x4000 pixels. To maintain the original aspect ratio of the image, we first centered crop the image to 8000x8000 and then resized them to 4000x4000. The images which were around 4000 were used directly after centered crop to 4000x4000.

B. Model Selection

We performed various experiments with different resolutions and batch sizes using early stop method. We found that 4000x4000 size is good considering objective of this work and accuracy. Due to GPU memory constraint, we need to keep the batch size of 16 per device with this resolution as compared to 32 in original paper. Rest of the hyper-parameters were kept same as the original model.

C. Optimization Procedure

We employed various techniques for optimizing training and inference of the model.

1) **Data Loading using multiple I/O process:**
We used optimal number of workers i.e. 2 (3 I/O process) in PyTorch *DataLoader* to reduce the data loading time.

2) **Distributed Learning:**
To make training process more efficient, we used *DistributedDataParallel* from PyTorch, to train the model across 4 GPUs on HPC. *DistributedSampler* was used to divide the training and validation set across 4 GPUs. For distributed process, we employed the learning rate policy as described in [2] i.e. we kept small fixed learning rate for first 5 epochs and afterwards we used linear scaling for learning rate along with Cosine Annealing schedule.

3) **Just In Time Compilation:**
We used *torch.jit.trace()* from PyTorch to trace the model to ScriptedModule which traces all the operations performed by model on input tensor and build computation graph and optimizes it during compilation. After few changes in original model, we were able to trace the model successfully.

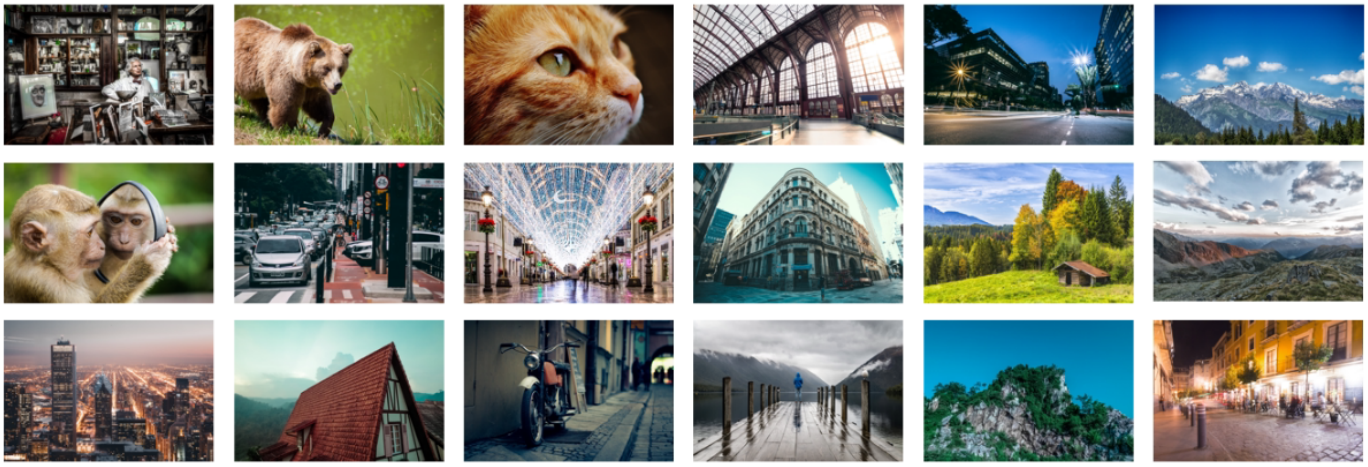

Fig. 4. Example training set images sampled from LIU4K

4) **Quantization:**
We employed post training quantization (PTQ) method using *torch.ao.quantization* submodule of PyTorch for static quantization and *torch.quantization.quantize_dynamic* for dynamic quantization to reduce model size and to make inference much faster.

D. Profiling tools and methods

To profile above techniques and to measure how well they are performing, we used Weight&Bases, PyTorch profiler and python *time* module.

With the help of Weight&Biases, it was very easy to compare performance (loss, accuracy, time) of various of experiments. It also captures the system metrics like GPU memory usage, GPU Utilization, etc.

PyTorch profiler helps in analyzing time spent in various GPU kernels and thus you can optimize that specific part of your model. In case of Distributed, it helps to visualize time spent in computation and communication respectively.

We used *time* module to measure inference time and data loading, training and running time during training.

E. Evaluation Metrics

We calculated the speedup using following equation to measure the performance of distributed training.

$$\text{SpeedUp} = \frac{t_{\text{serial}}}{t_{\text{parallel}}}$$

For measuring performance of quantization, we simply compared accuracy and time before and after quantization.

IV. Experimental Results

A. Experimental Setup

For our work, we used NYU HPC platform to train the model with following specifications:

1) Number of Servers: 1
2) Number of GPUs: 4
3) CPU cores per task: 16
4) Memory: 128 GB
5) GPU: RTX8000
6) Framework: PyTorch

B. Performance Comparison

TABLE I
Comparison of Training Times

Per Epoch	Single Device	Distributed (4 GPUs)	SpeedUp
Data Loading	1000 secs	271 secs	3.7
Training	160 secs	90 secs	1.78
Running	1170 secs	360 secs	3.25

```
Eager Model Inference (CPU)
Inference Time per Batch: 3.696291569620371 secs

Graph Model Inference (CPU)
Inference Time per Batch: 3.6160579593852162 secs

Eager Model Inference (GPU)
Inference Time per Batch: 0.8330167206004262 secs

Graph Model Inference (GPU)
Inference Time per Batch: 1.9260910218581557 secs
```

Fig. 5. Inference Time Comparison

```
TorchScript Model Performance:

C++ Inference Time (CPU) per Image: 0.242967 secs
Python Inference Time (CPU) per Image: 0.24820535443723202 secs
```

Fig. 6. TorchScript Model Performance (CPU)

Graphs showing performance of various experiments.

```
TorchScript Model Performance:

C++ Inference Time (GPU) per Image: 1.11081 secs
Python Inference Time (GPU) per Image: 1.096297555603087 secs
```

Fig. 7. TorchScript Model Performance (GPU)

```
Accuracy:
Accuracy before Quantization = 0.5961538461538461; 155/260
Accuracy after Quantization (Static) = 0.5576923076923077; 145/260
Accuracy after Quantization (Dynamic) = 0.5961538461538461; 155/260
```

Fig. 8. Quantized Model Comparison (CPU)

```
Per Image
Eager Model Inference (CPU)
Mountain
Inference Time per Image: 0.14683712180703878 secs
Size (MB): 7.042413

Quantized (static) Model Inference (CPU)
Mountain
Inference Time per Image: 0.07806775439530611 secs
Size (MB): 1.972793

Quantized (dynamic) Model Inference (CPU)
Mountain
Inference Time per Image: 0.1689505334943533 secs
Size (MB): 5.822403

Graph Model Inference (CPU)
Model Successfully compiled/traced to ../saved_model/torchscript_hct_1.pth
Further predictions will be made using traced module
Inference Time per Image: 0.2401146134361624 7 secs

Graph + Quantized (static) Model Inference (CPU)
Model Successfully compiled/traced to ../saved_model/torchscript_quantized_hct.pth
Further predictions will be made using traced module
Inference Time per Image: 0.15679329447448254 secs

Graph + Quantized (dynamic) Model Inference (CPU)
Model Successfully compiled/traced to ../saved_model/torchscript_quantized_hct.pth
Further predictions will be made using traced module
Inference Time per Image: 0.2580686416476965 secs
```

Fig. 9. Quantized Model Performance (CPU)

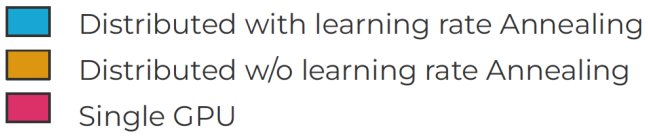

Fig. 10. Legend for below graphs

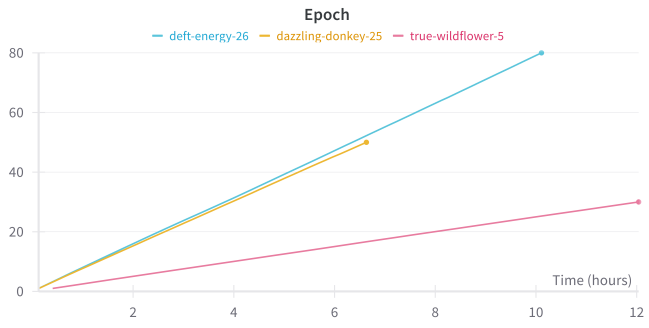

Fig. 11. Total Training Time

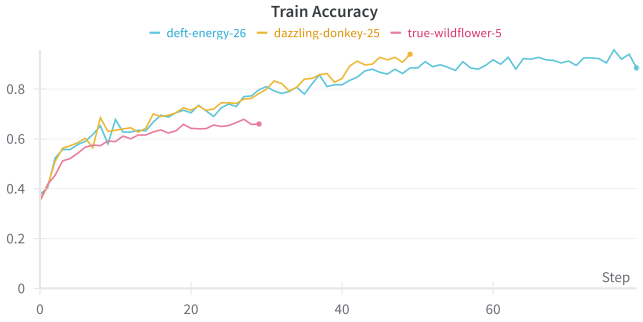

Fig. 12. Training Accuracy

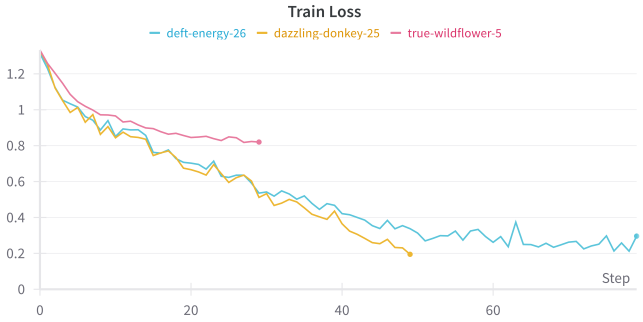

Fig. 13. Training Loss

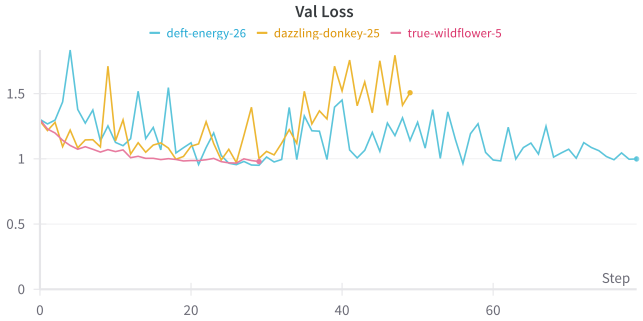

Fig. 14. Validation Loss

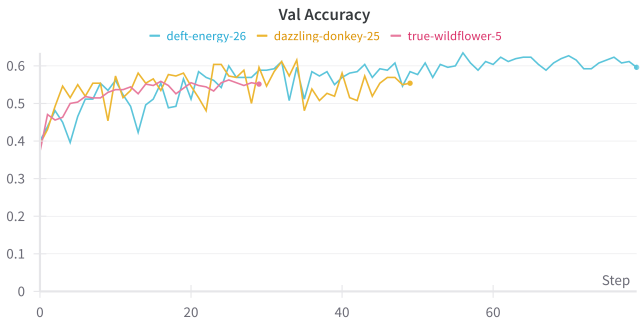

Fig. 15. Validation Accuracy

C. Analysis of Results

Figure 4 shows that we were able to achieve a overall speed of 3.25 with the help of Distributed Training across 4 GPUs.

Figure 5 shows that we were able to achieve a speed of 1.02 with the help of TorchScript on CPU.

Figure 6 and 7 shows that TorchScript performance is similar in python and non-python (C++) environment for both CPU and GPU.

Figure 8 and 9 shows that statically quantized the model results in 71.9% decreased model size, and 1.9x better inference time, but only 3.8% loss in accuracy. While dynamically quantized model results in 17.3% decreased model size, 1.01x inference time and 0% loss in accuracy.

In figure 14 and 15, we can see that with the help of learning rate annealing schedule, we were able to get consistent validation result in comparison to w/o annealing.

V. Discussion

A. Interpretation of Results

- Able to achieve very good speedup using distributed training over single device.
- Model was successfully traced using torch.jit.trace() and hence can be deployed on various machines without python environment.
- Efficiency of inference of Single image is better on CPU in comparison with GPU using TorchScripted model.
- Reduced model size by a huge amount with the help of post training quantization.
- Inference time using C++ and CPU per image is less and thus we can efficiently deploy the model on machines without python and GPU.

B. Comparison with Previous Studies

In original paper [1], they were able to achieve an accuracy of 90% on medical images. We were not able to achieve same validation accuracy due to two main reasons:
1. Lack of data: Only 1600 training images which is very less to train neural network.
2. Original model was designed specifically for medical images but since we were not able to access original dataset, we need to train on different dataset.

But we were able to achieve the objectives within our scope successfully i.e. optimizing the training and inference process of HCT model.

C. Challenges and Limitations

The primary obstacle we encountered revolves around data scarcity. Existing datasets containing high-resolution images are notably limited in quantity, and those available offer only a sparse number of samples. This dearth of data presents a significant challenge, particularly when training transformer models, as the insufficiency of samples impedes the model's ability to generalize effectively.

Secondly, there is a limited support and documentation in PyTorch for post training quantization. It does not support GPU as of now.

D. Future Directions

The primary task in future will be to create a better and consistent public dataset for High Resolution Images. It will help to achieve to better validation accuracy.

We can also further scale the training process by increasing number of GPUs and by experimenting various gradients synchronization methods like K Sync SGD, K-batch Sync SGD, K-Async SGD as mentioned in [4].

We can improve accuracy of static quantization with fusing, quant aware training (QAT), and partial quantization (meaning we only quantize the layers with largest size in bytes), with different percent of partial quantization, hence we can trade-off model size and inference time for accuracy.

VI. Conclusion

A. Summary of Findings

Training the model using DistributedDataParallel helps to achieve very good speedup for training process.

Quantization helps to reduce the model by nearly 80% which is a great savings in term of memory because many edge devices do not have much memory.

PyTorch's trace() utility is still not optimized for all types models like in our case, we were not able to get speedup using trace() on GPU.

B. Contributions

- Data Pre-processing - Ankit
- Basic Training - Gaurav
- Distributed Training - Ankit
- Just In Time Compilation - Ankit
- Quantization - Gaurav

C. Recommendations for Future Research

Original model was proposed for classifying medical images but we used the model for classification of natural scenes. More research is needed to adapt this model to other tasks like satellite image analysis and any area where high resolution images have high importance.

References

[1] A. Taha, Y. N. Truong Vu, B. Mombourquette, T. P. Matthews, J. Su, and S. Singh, "Deep is a luxury we don't have," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 25–35, Springer, 2022.

[2] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[3] J. Liu, D. Liu, W. Yang, S. Xia, X. Zhang, and Y. Dai, "A comprehensive benchmark for single image compression artifacts reduction," in *arXiv*, 2019.

[4] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar, "Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd," in *International conference on artificial intelligence and statistics*, pp. 803–812, PMLR, 2018.