Article | June 2022

Computer Vision, Speech and Natural Language Processing

Deploying Transformers on the Apple Neural Engine

An increasing number of the machine learning (ML) models we build at Apple each year are either partly or fully adopting the Transformer architecture ↗. This architecture helps enable experiences such as panoptic segmentation in Camera with HyperDETR ›, on-device scene analysis in Photos ›, image captioning for accessibility ›, machine translation ›, and many others. This year at WWDC 2022, Apple is making available an open-source reference PyTorch ↗ implementation of the Transformer architecture, giving developers worldwide a way to seamlessly deploy their state-of-the-art Transformer models on Apple devices.

This implementation is specifically optimized for the Apple Neural Engine (ANE), the energy-efficient and high-throughput engine for ML inference on Apple silicon. It will help developers minimize the impact of their ML inference workloads on app memory, app responsiveness, and device battery life. Increasing the adoption of on-device ML deployment will also benefit user privacy, since data for inference workloads remains on-device, not on the server.

In this article we share the principles behind this reference implementation to provide generalizable guidance to developers on optimizing their models for ANE execution. Then, we put these principles into action and showcase how to deploy an example pretrained Transformer model, the popular Hugging Face distilbert ↗, in just a few lines of code. Notably, this model, which works out-of-the-box and on device using Core ML already, is up to 10 times faster and consumes 14 times less memory after our optimizations.

The Transformer Architecture

Published in 2017, the Transformer ↗ architecture has had a great impact in many fields, including natural language processing and computer vision, in a short period of time. Models built on this architecture produce state-of-the-art results across a wide spectrum of tasks while incurring little to no domain-specific components. This versatility has resulted in a proliferation of applications built on this architecture.

Most notably, the Hugging Face model hub ↗ hosts tens of thousands of pretrained Transformer models, such as variants of GPT-2 ↗ and BERT ↗, which were trained and shared by the ML community. Some of these models average tens of millions of monthly downloads, contributing to the research momentum behind training bigger, better, and increasingly multitask Transformer models and giving birth to the development of efficient deployment strategies on both the device side ↗ and the server side ↗.

The Apple Neural Engine

The first generation of the Apple Neural Engine (ANE) was released as part of the A11 chip found in iPhone X, our flagship model from 2017. It had a peak throughput of 0.6 teraflops (TFlops) in half-precision floating-point data format (float16 or FP16), and it efficiently powered on-device ML features such as Face ID and Memoji.

Fast-forward to 2021, and the fifth-generation of the 16-core ANE is capable of 26 times the processing power, or 15.8 TFlops, of the original. Since 2017, use of the ANE has been steadily increasing from a handful of Apple applications to numerous applications from both Apple and the developer community. The availability of the Neural Engine also expanded from only the iPhone in 2017 to iPad starting with the A12 chip and to Mac starting with the M1 chip.

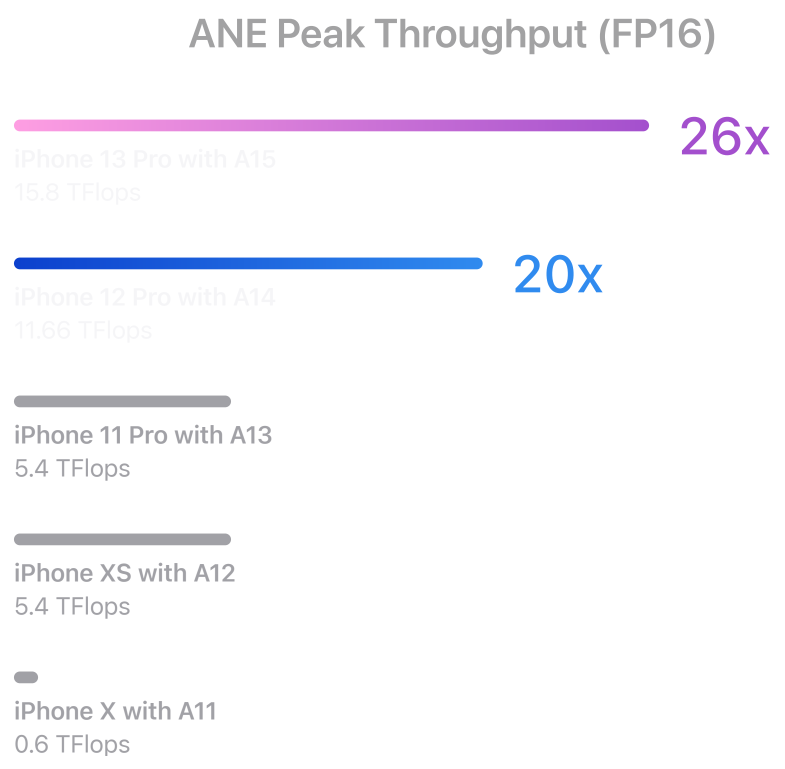

Figure 1: The evolution of the Apple Neural Engine, 2017 to 2021. The 16-core Neural Engine on the A15 Bionic chip on iPhone 13 Pro has a peak throughput of 15.8 teraflops, an increase of 26 times that of iPhone X.

When training ML models, developers benefit from accelerated training on GPUs with PyTorch ↗ and TensorFlow › by leveraging the Metal Performance Shaders (MPS) back end. For deployment of trained models on Apple devices, they use coremltools ↗, Apple's open-source unified conversion tool, to convert their favorite PyTorch and TensorFlow models to the Core ML model package format. Core ML then seamlessly blends CPU, GPU, and ANE (if available) to create the most effective hybrid execution plan exploiting all available engines on a given device. It lets a wide range of implementations of the same model architecture benefit from the ANE even if the entire execution cannot take place there due to idiosyncrasies of different implementations. This workflow is designed to make it easy for developers to deploy models on Apple devices without having to worry about the capabilities of any particular device or implementation.

Principles Behind Optimizing Transformers for the Neural Engine

Although the implementation flexibility that hybrid execution offers is simple and powerful, we can trade this flexibility off in favor of a particular and principled implementation that deliberately harnesses the ANE, resulting in significantly increased throughput and reduced memory consumption. Other benefits include mitigating the inter-engine context-transfer overhead and opening up the CPU and the GPU to execute non-ML workloads while ANE is executing the most demanding ML workloads.

In this section, we describe some of the generalizable principles behind our reference implementation for Transformers with the goal of empowering developers to optimize models they intend to deploy on the ANE.

Principle 1: Picking the Right Data Format

In general, the Transformer architecture processes a 3D input tensor that comprises a batch of B sequences of S embedding vectors of dimensionality C. We represent this tensor in the (B, C, 1, S) data format because the most conducive data format for the ANE (hardware and software stack) is 4D and channels-first.

The native torch.nn.Transformer ↗ and many other PyTorch implementations use either the (B, S, C) or the (S, B, C) data formats, which are both channels-last and 3D data formats. These data formats are compatible with nn.Linear layers ↗, which constitute a major chunk of compute in the Transformer. To migrate to the desirable (B, C, 1, S) data format, we swap all nn.Linear ↗ layers with nn.Conv2d layers ↗. Furthermore, to preserve compatibility with previously trained checkpoints using the baseline implementation, we register a load_state_dict_pre_hook ↗ to automatically unsqueeze the nn.Linear weights twice in order to match the expected nn.Conv2d weights shape as shown here ↗.

The mapping of the sequence (S) axis to the last axis of the 4D data format is very important because the last axis of an ANE buffer is not packed; it must be contiguous and aligned to 64 bytes. This constraint applies only to the last axis, and the ANE compiler determines whether the rest of the axes are packed for best performance. Although unpacked buffers allow faster read times, if used improperly, they cause larger than necessary buffers to be allocated. For example, if the last axis is used as a singleton one by the model implementation's data format, it will be padded to 64 bytes, which results in 32 times the memory cost in 16-bit and 64 times the memory cost in 8-bit precision. Such an increase in buffer size will significantly reduce the chance of L2 cache residency and increase the chance of hitting DRAM. This is not desirable from a power and latency perspective.

Principle 2: Chunking Large Intermediate Tensors

For the multihead attention function in the Transformer, we split ↗ the query, key, and value tensors to create an explicit list of single-head attention functions, each of which operates on smaller chunks of input data. Smaller chunks increase the chance of L2 cache residency as well as increasing multicore utilization during compilation.

Principle 3: Minimizing Memory Copies

Given that at least one axis of ANE buffers is not packed, reshape and transpose operations are likely to trigger memory copies unless specifically handled. In our reference implementation, we avoid all reshapes and incur only one transpose ↗ on the key tensor right before the query and key matmul. We avoid further reshape and transpose operations during the batched matrix multiplication operation for scaled dot-product attention by relying on a particular einsum operation. Einsum is a notational convention that implies summation over a set of indexed terms in a formula. We use the bchq,bkhc->bkhq einsum formula ↗, which represents a batched matmul operation whose data format directly maps to hardware without intermediate transpose and reshape operations.

Principle 4: Handling Bandwidth-Boundness

Even after all the optimizations, many Transformer configurations become bandwidth-bound on the ANE when the sequence length is relatively short. This is due to the fact that large parameter tensors are being fetched from memory, only to be applied on too few inputs before the next parameter tensor is fetched. Fetching from memory dominates overall latency in these cases. Bandwidth-boundness manifests very clearly in Figure 2. The optimized model's latency stays approximately constant across sequence lengths of 32, 64, and 128 (with batch size 1) even though the computational load quadruples. One way to escape the bandwidth-bound regime is to increase the batch size for batch inference workloads. Another way is to reduce the parameter tensor size by quantization or pruning such that memory fetching becomes cheaper and faster.

Principles Packaged into Code: ane_transformers

We share the reference implementation built on these principles and distribute it on PyPI to accelerate Transformers running on Apple devices with an ANE, on A14 and later or M1 and later chips. The package is called ane_transformers and the first on-device application using this package was HyperDETR ›, as described in our previous article.

Next, we showcase the application of these principles to a pretrained Transformer model: distilbert from Hugging Face ↗. Code for this example is also made available through ane_transformers.

Case Study: Hugging Face distilbert

As the scaling laws of deep learning continue to hold, the ML community is training bigger and more capable Transformer models. However, most open-source implementations of the Transformer architecture are optimized for either large-scale training hardware or no hardware at all. Furthermore, despite significant concurrent advancements in model compression, the state-of-the-art models are scaling faster than compression techniques are improving.

Hence, there is immense need for on-device inference optimizations to translate these research gains into practice. These optimizations will enable ML practitioners to deploy much larger models on the same input set, or to deploy the same models to run on much larger sets of inputs within the same compute budget.

In this spirit, we take the principles behind our open-source reference PyTorch implementation for the Transformer architecture and apply them to the popular distilbert model from the Hugging Face model hub in a case study ↗. Applying these optimizations resulted in a forward pass that is up to 10 times faster with a simultaneous reduction of peak-memory consumption of 14 times on iPhone 13. Using our reference implementation, on a sequence length of 128 and a batch size of 1, the iPhone 13 ANE achieves an average latency of 3.47 ms at 0.454 W and 9.44 ms at 0.072 W. Even using our reference implementation, the ANE peak throughput is far from being saturated for this particular model configuration, and the performance may be further improved with using the quantization and pruning techniques as discussed earlier.

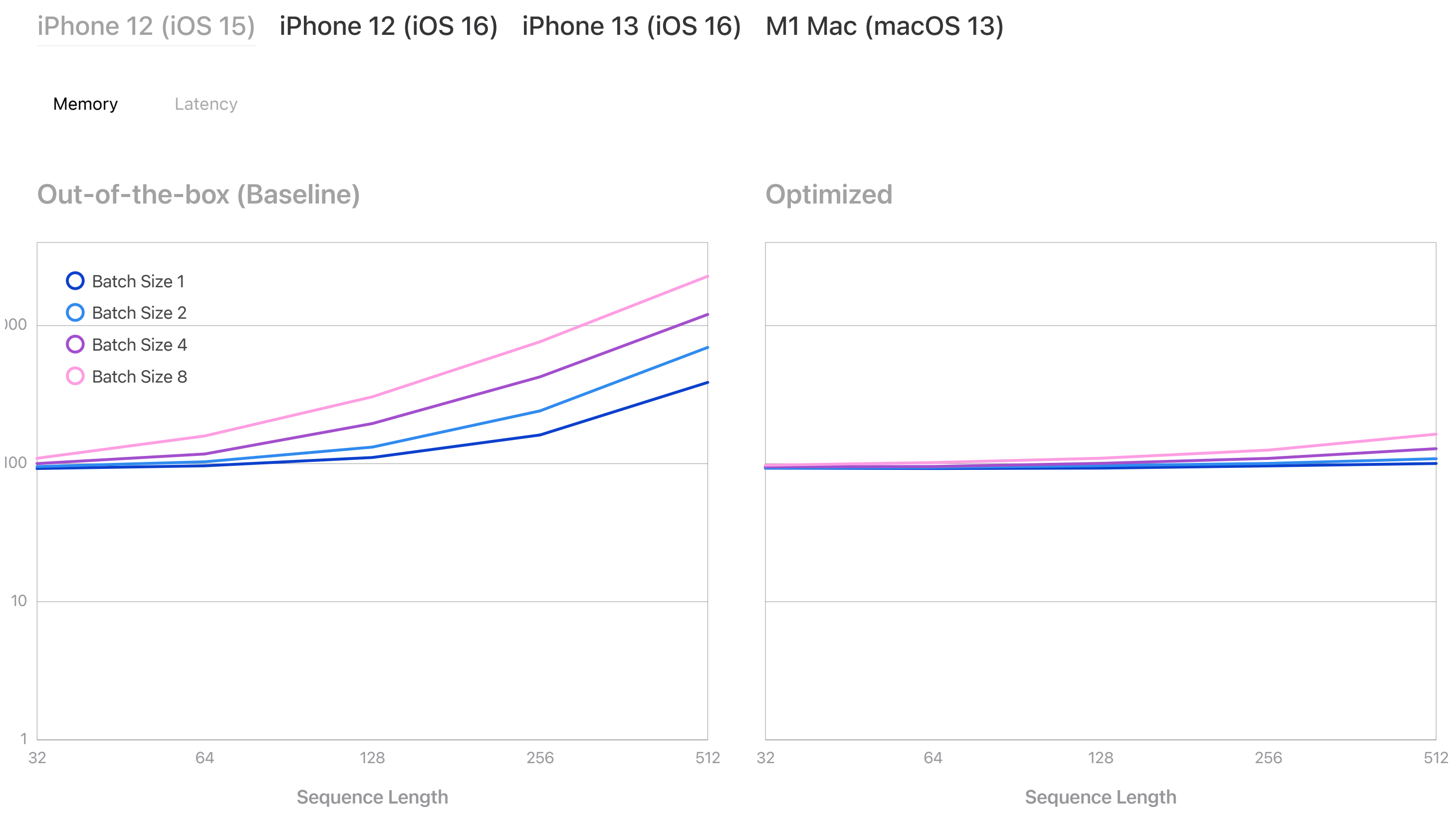

Figure 2: Performance curves (latency, memory) for Hugging Face distilbert on various Apple devices and operating system versions. Shaded ranges indicate performance under varying power consumption.

To contextualize the numbers we just reported, a recent article ↗ from Hugging Face and AWS reported "the average latency . . . is 5-6 ms for a sequence length of 128" for the same model in our case study, when deployed on the server side using ML-optimized ASIC hardware ↗ from AWS. Based on this external data point, we are happy to report that our on-device inference latency compares favorably to those measured on an optimized server-side inference engine while executing on a device that is orders of magnitude more energy-constrained. Figure 2 shows the latency and memory consumption of the same model across different sequence lengths, batch sizes, and devices.

Putting It All Together: From PyTorch to Xcode

Finally, we show how these optimizations are applied through just a few lines of code, and then we profile the model in Xcode using the new Core ML Performance Report feature available in Xcode 14.

To begin, we initialize the baseline distilbert model ↗ from the Hugging Face model hub:

```
import transformers
model_name = "distilbert-base-uncased-finetuned-sst-2-english"
baseline_model = transformers.AutoModelForSequenceClassification.from_pretrai
    model_name,
    return_dict=False,
    torchscript=True,
).eval()
```

Then we initialize the optimized model, and we restore its parameters using that of the baseline model to achieve high output parity as measured by peak-signal-to-noise-ratio (PSNR):

```
from ane_transformers.huggingface import distilbert as ane_distilbert
optimized_model = ane_distilbert.DistilBertForSequenceClassification(
    baseline_model.config).eval()
optimized_model.load_state_dict(baseline_model.state_dict())
```

Next we create sample inputs for the model:

```
tokenizer = transformers.AutoTokenizer.from_pretrained(model_name)
tokenized = tokenizer(
    ["Sample input text to trace the model"],
    return_tensors="pt",
    max_length=128,  # token sequence length
    padding="max_length",
)
```

We then trace the optimized model to obtain the expected model format (TorchScript) ↗ for the coremltools ↗ conversion tool.

```
import torch
traced_optimized_model = torch.jit.trace(
    optimized_model,
    (tokenized["input_ids"], tokenized["attention_mask"])
)
```

Finally, we use coremltools to generate the Core ML model package file and save it.

```
import coremltools as ct
import numpy as np
ane_mlpackage_obj = ct.convert(
    traced_optimized_model,
    convert_to="mlprogram",
    inputs=[
        ct.TensorType(
                f"input_{name}",
                    shape=tensor.shape,
                    dtype=np.int32,
                ) for name, tensor in tokenized.items()
            ],
)
out_path = "HuggingFace_ane_transformers_distilbert_seqLen128_batchSize1.mlpa
ane_mlpackage_obj.save(out_path)
```

To verify performance, developers can now launch Xcode and simply add this model package file as a resource in their projects. After clicking on the Performance tab, the developer can generate a performance report on locally available devices, for example, on the Mac that is running Xcode or another Apple device that is connected to that Mac. Figure 3 shows a performance report generated for this model on an iPhone 13 Pro Max with iOS 16.0 installed.

Xcode Core ML Performance Report

Baseline Optimized

Figure 3: Core ML performance reports generated in Xcode. Developers can review runtime statistics, layer dispatch to each engine, and accomplish additional tasks.

Conclusion

Transformers are becoming ubiquitous in ML as their capabilities scale up with their size. Deploying Transformers on-devices requires efficient strategies, and we are thrilled to provide guidance to developers on this topic. Learn more about the implementation described in this post on the Machine Learning ANE Transformers GitHub. ↗

Acknowledgments

Many people contributed to this work, including Atila Orhon, Aseem Wadhwa, Youchang Kim, Francesco Rossi, and Vignesh Jagadeesh.

Resources

Apple Developer. “Framework: Core ML.” 2022. [link.] ›

Apple Developer. “Getting Started with tensorflow-metal PluggableDevice.” 2022. [link.] ›

Apple Developer. "Accelerate machine learning with Metal." 2022. [link.] ›

“Apple Neural Engine Transformers.” GitHub. 2022. [link.] ↗

Hugging Face. “Models.” 2022. [link.] ↗

PyTorch. “From Research to Production.” 2022. [link.] ↗

PyTorch. “Introducing Accelerated PyTorch Training on Mac.” 2022. [link.] ↗

“Trending Research.” Papers with Code. 2022. [link.] ↗

References

Apple. “On-Device Panoptic Segmentation for Camera Using Transformers.” Machine Learning Research, October 2021. [link.] ›

Apple. “Use VoiceOver for Images and Videos on iPhone.” iPhone User Guide. Cupertino, CA: Apple, 2022. [link.] ›

Core ML Tools. “Introduction: Use coremltools to Convert Models from Third-Party Libraries to Core ML.” 2022. [link.] ↗

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” 2019. [link.] ↗

OpenAI. “Better Language Models and Their Implications.” February 14, 2019. [link.] ↗

Schmid, Philipp. “Accelerate BERT Inference with Hugging Face Transformers and AWS Inferentia.” Hugging Face, 2022. [link.] ↗

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. “Attention Is All You Need.” 2017. [link.] ↗

Related readings and updates.

Stable Diffusion with Core ML on Apple Silicon

Today, we are excited to release optimizations to Core ML for Stable Diffusion ↗ in macOS 13.1 and iOS 16.2, along with code to get started with deploying to Apple Silicon devices.

See paper details

A Multi-Task Neural Architecture for On-Device Scene Analysis

Scene analysis is an integral core technology that powers many features and experiences in the Apple ecosystem. From visual content search to powerful memories marking special occasions in one's life, outputs (or "signals") produced by scene analysis are critical to how users interface with the photos on their devices. Deploying dedicated models for each of these individual features is inefficient as many of these models can benefit from sharing resources. We present how we developed Apple Neural Scene Analyzer (ANSA), a unified backbone to build and maintain scene analysis workflows in production. This was an important step towards enabling Apple to be among the first in the industry to deploy fully client-side scene analysis in 2016.

See article details

Discover opportunities in Machine Learning.

Our research in machine learning breaks new ground every day.

Work with us