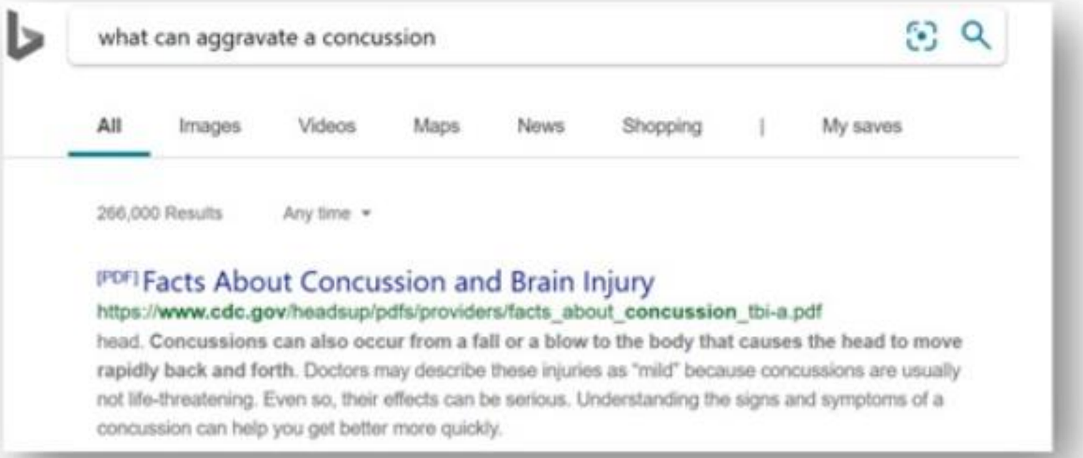
ONNX Runtime

Breakthrough optimizations for transformer inference on GPU and CPU

Emma Ning| Senior product manager, Microsoft

BEFORE



AFTER

Bing Search Engine

Using cutting-edge NLP techniques like transformers

to better understand user queries, webpages, and other documents

Transformer for natural language processing

- Transformers - breakthrough in natural language understanding
- BERT - Bidirectional Encoder Representations from Transformers
  - Architecture (L: layers (transformer block), H:hidden size, A: self-attention head)
    - BERT base: L=12, H=768, A=12, Parameters=110M
    - BERT large: L=24, H=1024, A=16, Parameters=340M
  - Running a 12- or 24-layer BERT for every query real-time is prohibitively expensive

BERT optimizations in Bing

Finetuned BERT-Base, a 12-layermodel

- Significantly improves Precision and Coverage

Leveraged knowledge distillation to create a 3-layered BERT

- No any significant loss in accuracy
- Reduced inference cost significantly
- Still benchmarked at 77ms serving latency on CPU

Re-implemented the model using TensorRT C++ APIs

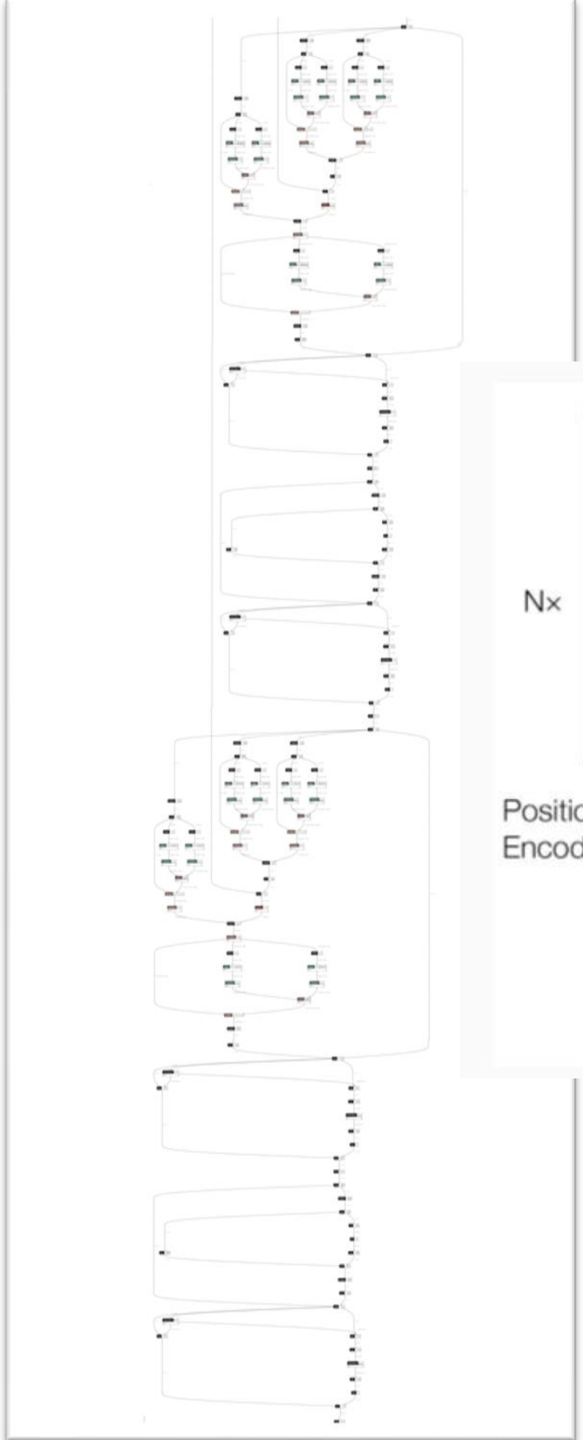
- Take full advantage of NVIDIA GPU architecture
- 800x throughput improvement on GPU

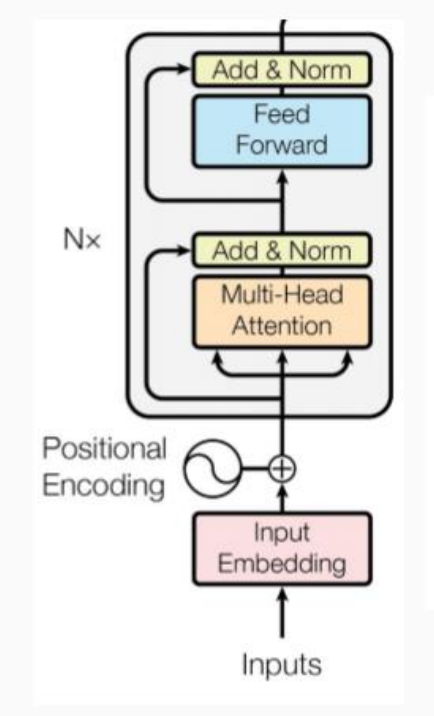
Problem - Reimplement the model was time-consuming

Transformer inference acceleration with ONNX Runtime

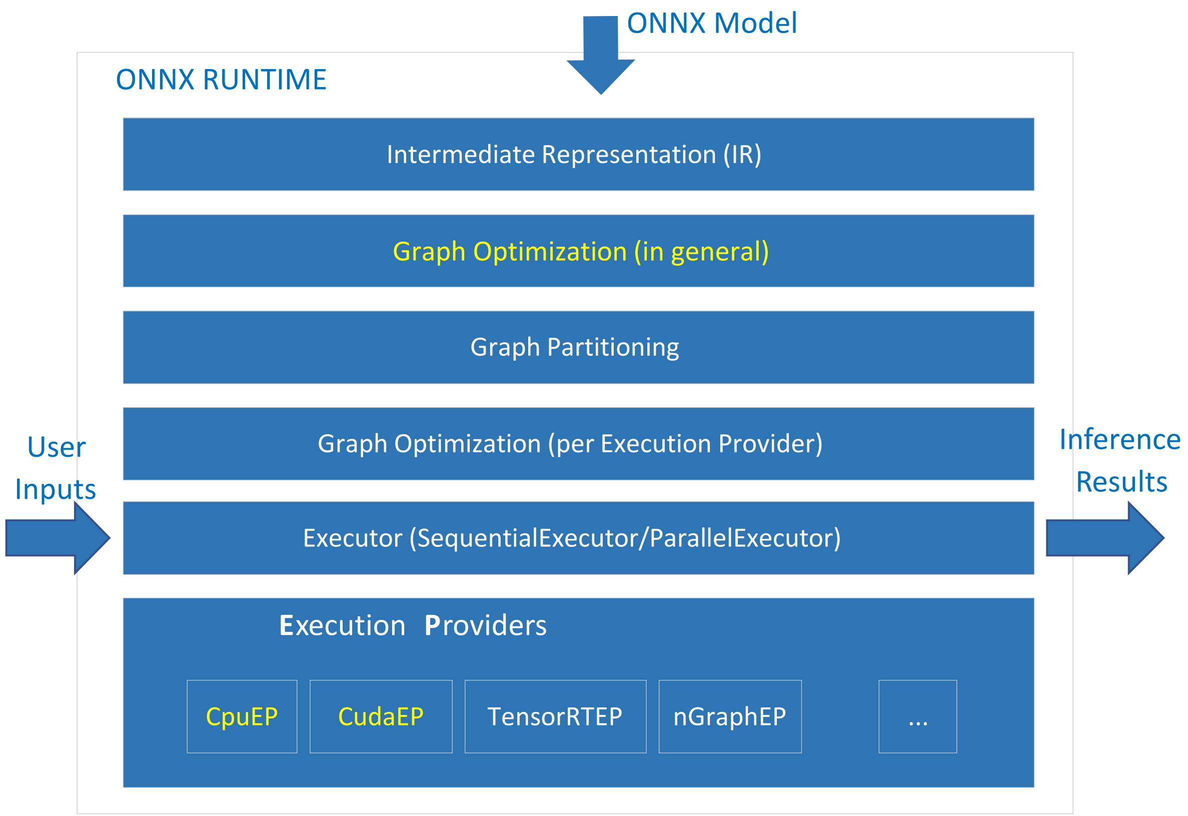
BERT Optimization Opportunity

- Model
  - Too many elementary operators
  - Multi transformer cells
- Kernels in ONNX Runtime
  - Not fully utilize hardware characteristic
    - CPU cores
    - Tensor-Core



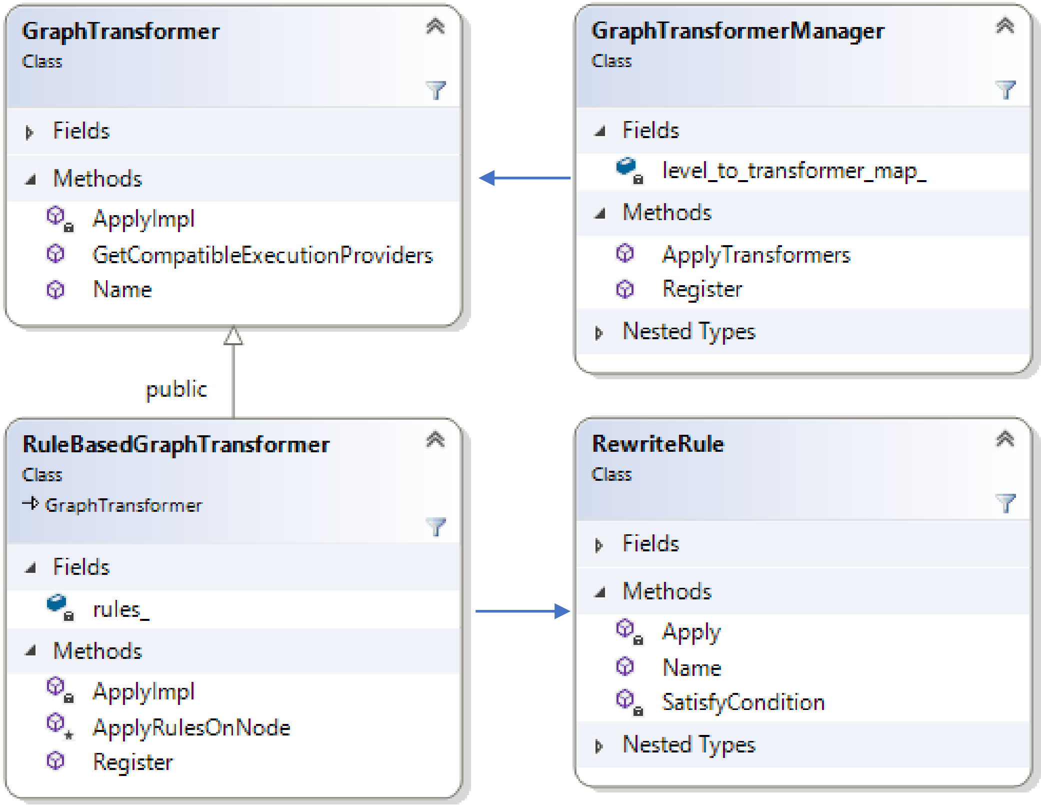


BERT optimization in ONNX Runtime



- **Graph optimization**
- Hardware-based kernel optimization

ONNX Runtime – Graph Optimization



GraphTransformer

- An interface created for finding patterns (with specific nodes) and applying rewriting rules against a sub-graph.
- An interface created for applying graph transformation with full graph editing capability.

Graph Optimization Level

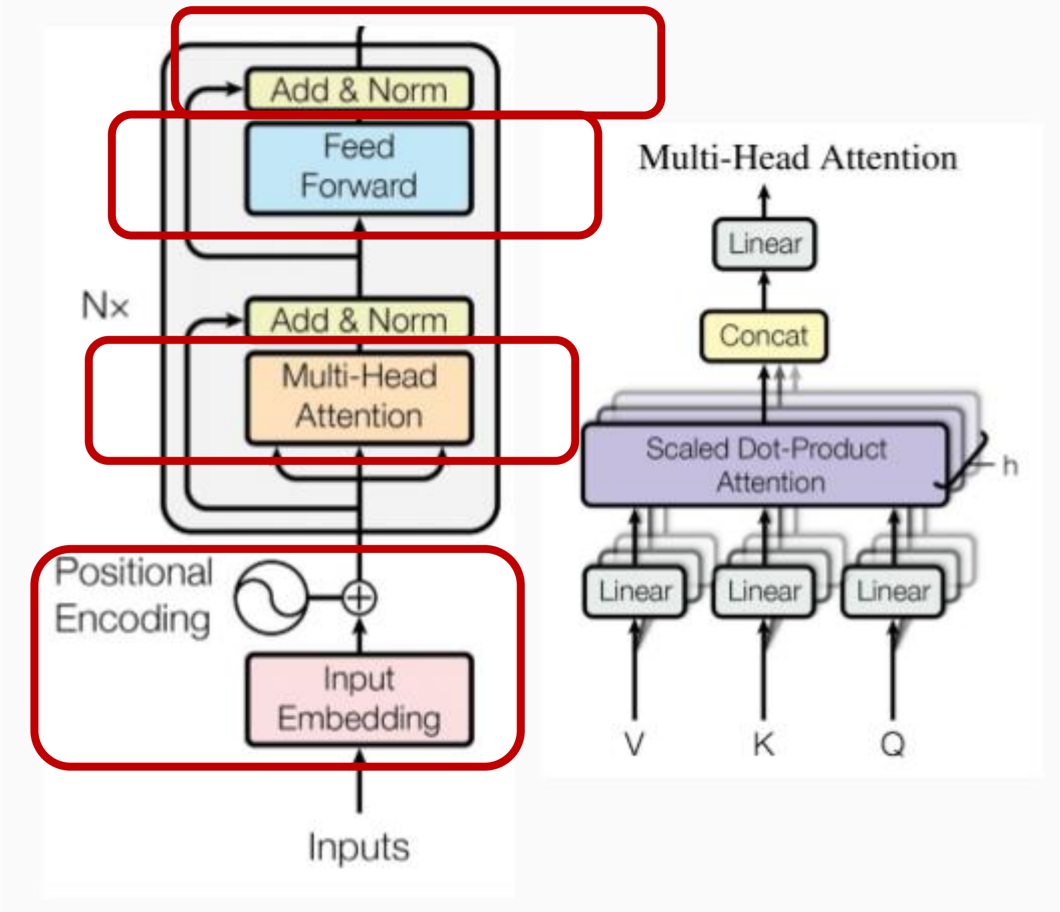
Basic: General transformers not specific to any specific execution provider (e.g. drop out elimination)

Extended: Execution provider specific transformers

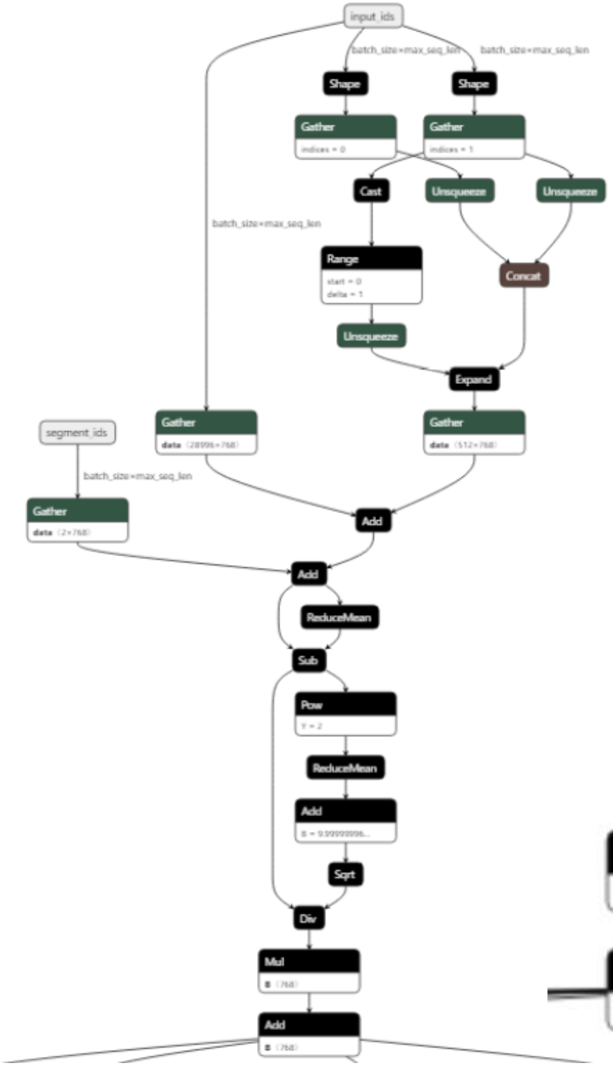
Layout Optimizations: change the data layout for applicable nodes (NCHW layout to NHWC layout)

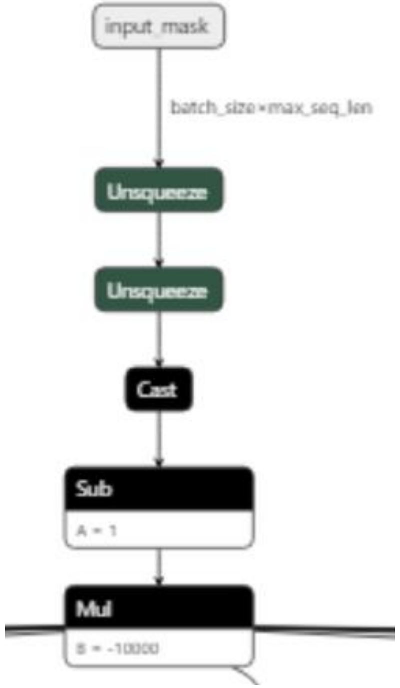
BERT Encoder Block

- Positional embeddings
- Multi-headed self-attention
- Feed-forward layers
- Layer norm and residuals

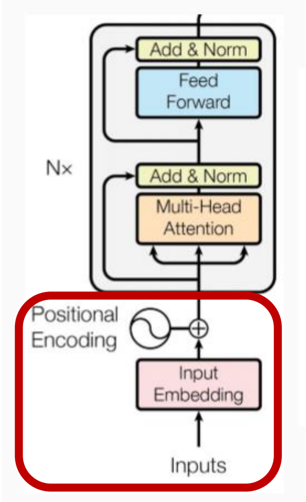


Embedding and Positional Encoding fusion

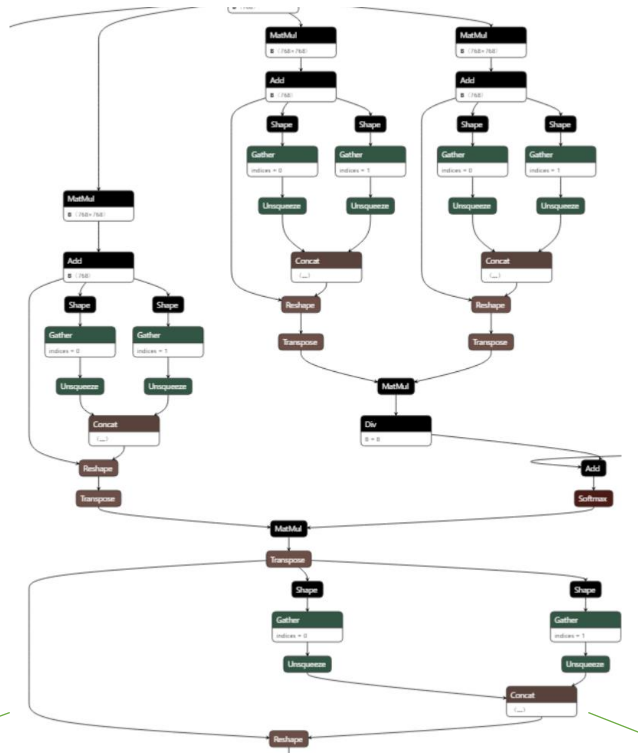


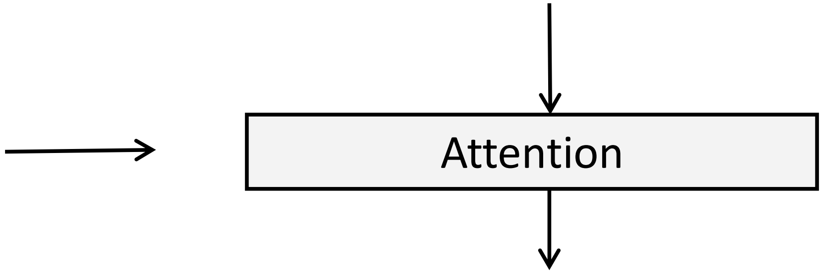


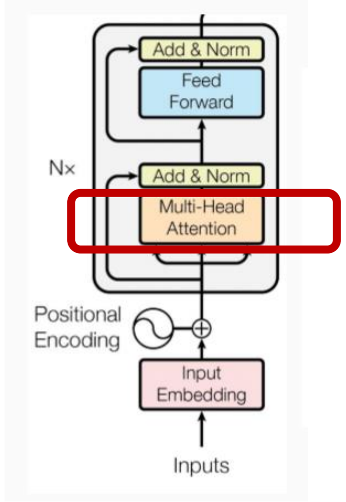
EmbedLayerNormalization

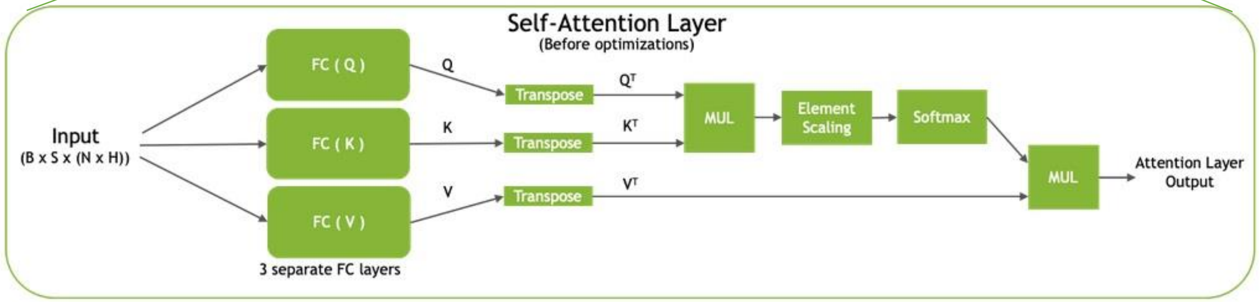


Multi-headed self-attention Fusion



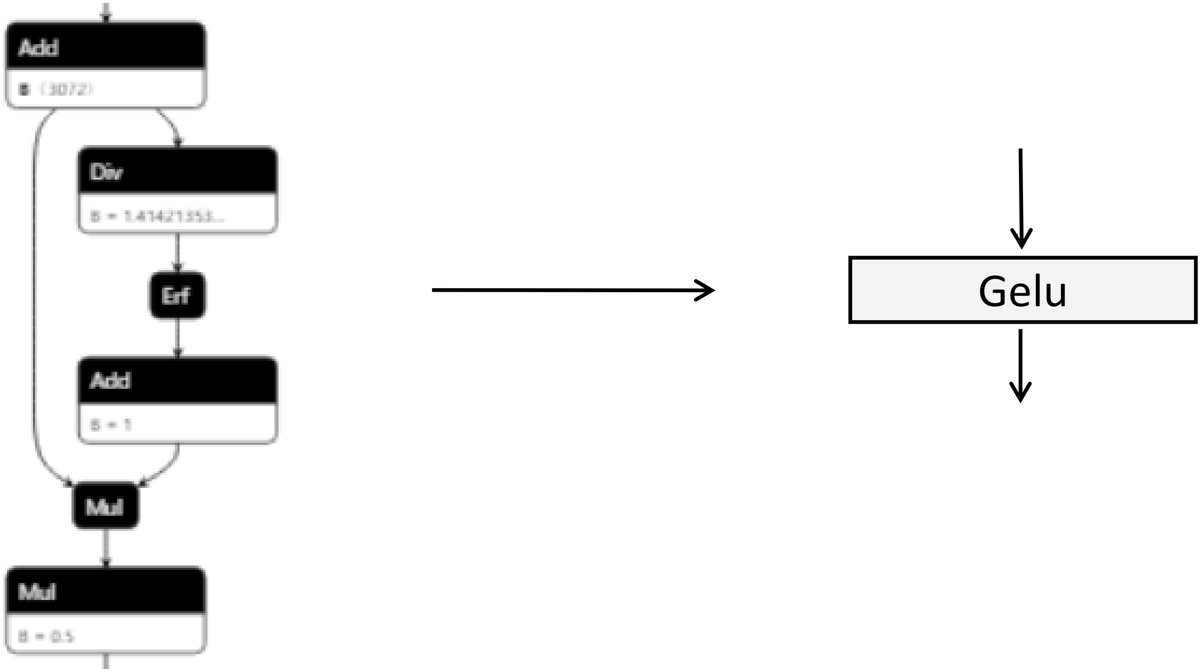


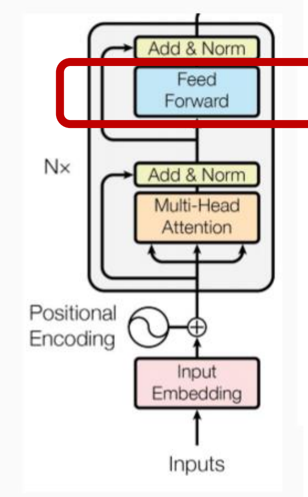




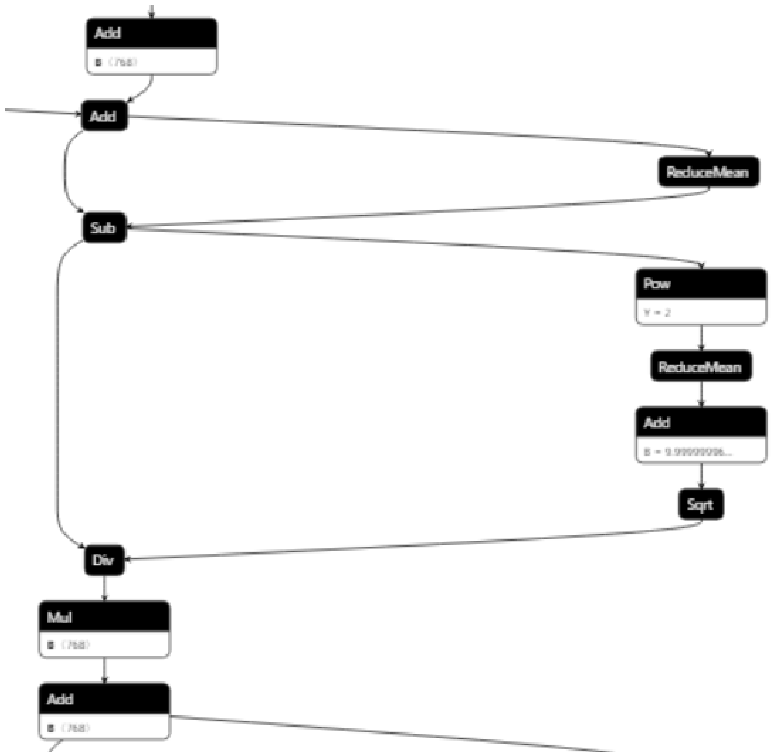
Gelu Fusion

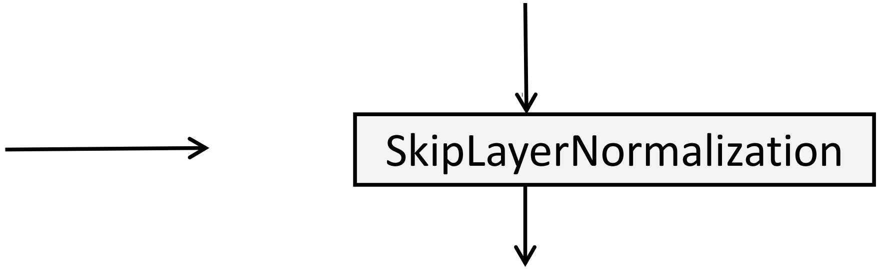
$$\mathrm{GELU}(x) := x\mathbb{P}(X \leq x) = x\Phi(x) = 0.5x\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

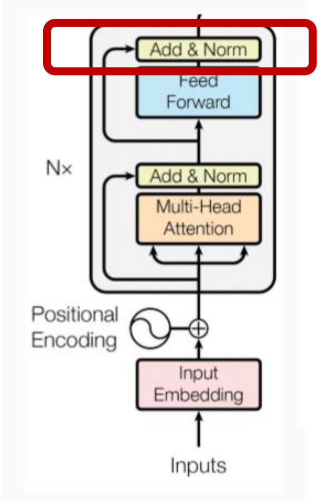




Skip Layer Normalization Fusion







BERT Graph Optimizations

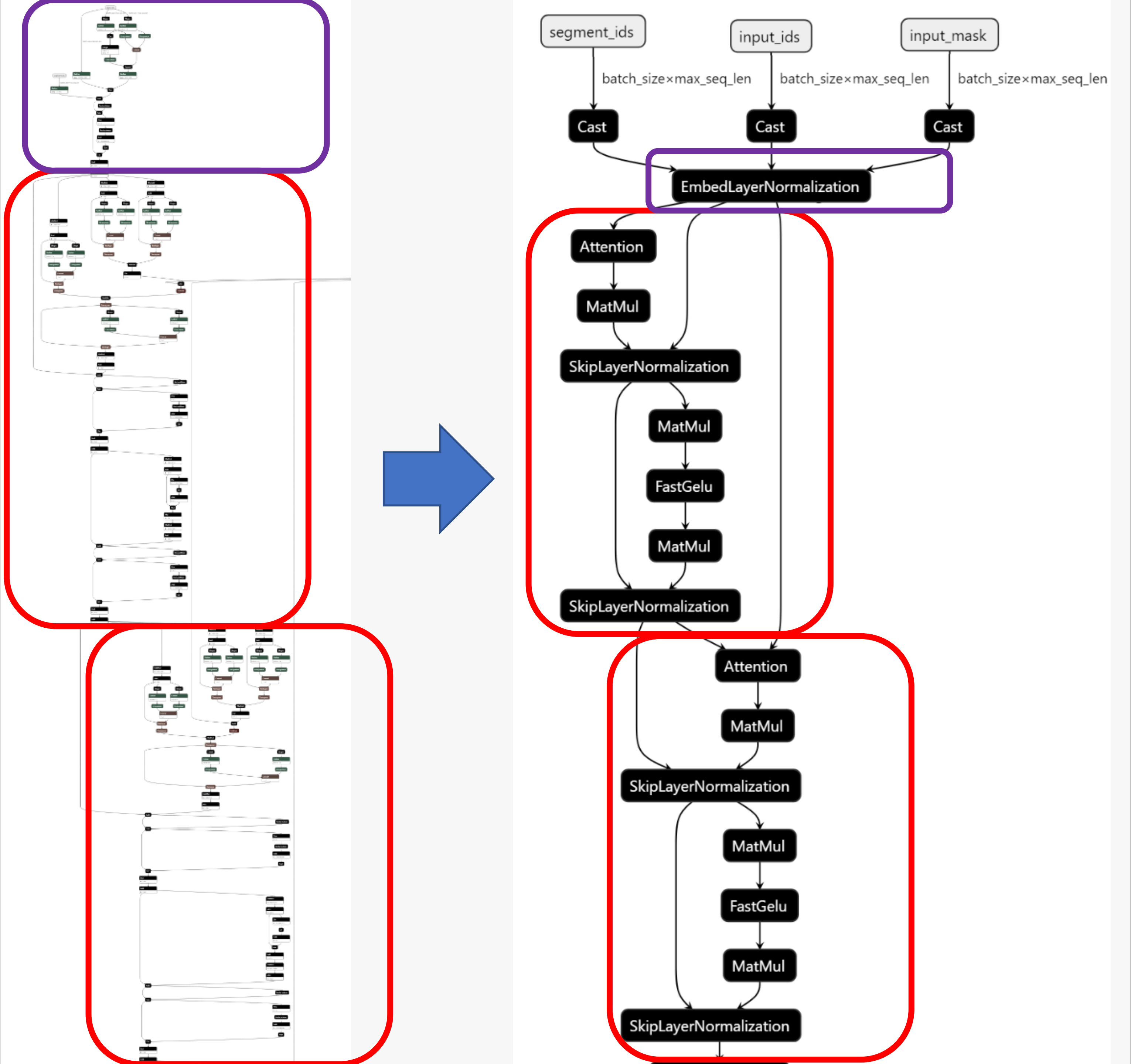
Basic Level

- Constant Folding
- Reshape Fusion

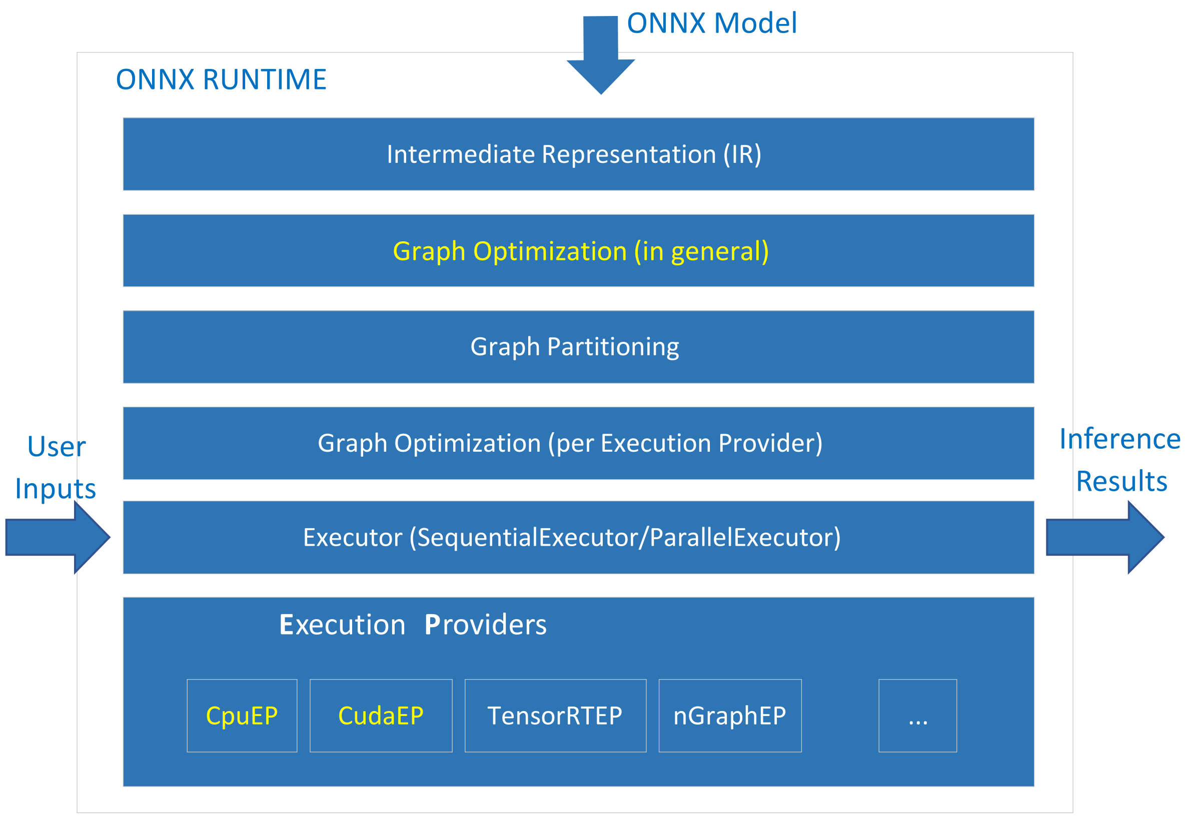
Extended Level

- GELU Fusion
- Layer Normalization Fusion
- BERT Embedding Layer Fusion
- Attention Fusion
- Skip Layer Normalization Fusion
- Bias GELU Fusion
- Fast GELU Fusion

After graph optimization



BERT optimization in ONNX Runtime



- Graph optimization
- **Hardware-based kernel optimization**
  - Optimized CPU and CUDA kernels in ONNX Runtime
    - Take full advantage of the GPU architecture
    - In self-attention layer CPU implementation
      - Increase the parallelization and fully leverage available CPU cores
      - Leverage GEMM to further reduce the computation cost

BERT With ONNX Runtime

BERT-SQUAD with 128 sequence length and batch size 1 on Azure Standard NC6S_v3 (GPU V100)

- in 1.7 ms for 12-layer fp16 BERT-SQUAD.
- in 4.0 ms for 24-layer fp16 BERT-SQUAD.

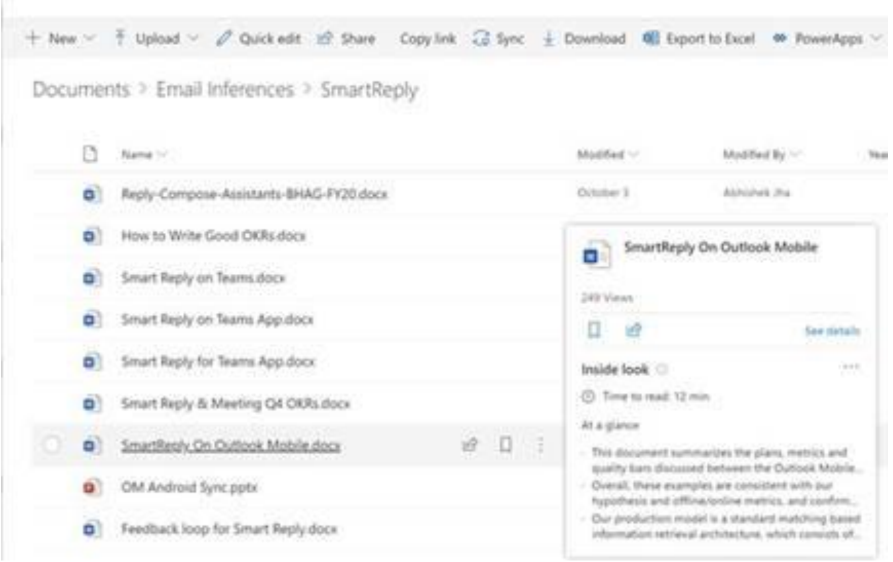
Bing's 3-layer BERT with 128 sequence length

		Batch size	Inference on	Throughput (Query per second)	Latency (milliseconds)
CPU	Original 3-layer BERT	1	Azure Standard F16s_v2 (CPU)	6	157
	ONNX Model	1	Azure Standard F16s_v2 (CPU) **with ONNX Runtime**	111	9
GPU	Original 3-layer BERT	4	Azure NV6 GPU VM	200	20
	ONNX Model	4	Azure NV6 GPU VM **with ONNX Runtime**	500	8
	ONNX Model	64	Azure NC6S_v3 GPU VM **with ONNX Runtime + System Optimization** (Tensor Core with mixed precision, Same Accuracy)	10667	6

On NVIDIA V100 GPUs we saw ~10,000 queries per second throughput

Development time for new BERT scenarios was cut from multiple days to a few hours

ONNX Runtime powered BERT inference in office



"At a glance" in In **OneDrive** and **Sharepoint**

Key points In **Word**

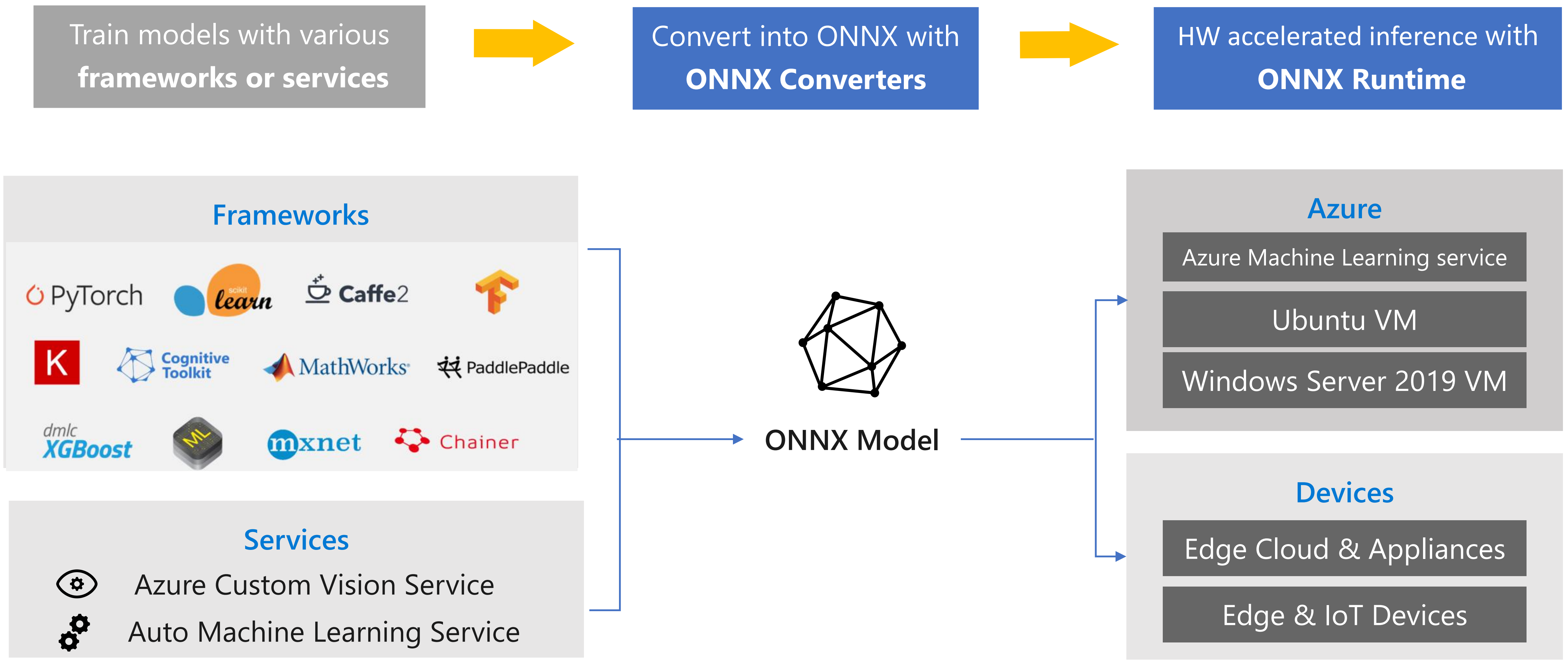
Keypoints model

- 3-layer BERT
- The P50 latency reduced by 3x over the original traditional ML based solution
- The development cost was significantly reduced

ONNX Runtime to power BERT inference

Bing Ranking **3-layer BERT**	**Key Point in Office** **3-layer BERT**	**Bing Ads** **3-layer BERT**
Hugging Face **12-layer BERT**	**Text Analytics in Azure AI** **12-layer BERT**	**Bing Feeds** **12-layer BERT**
Speech & Language in Azure Cognitive service **2-layer BERT**	**Questions suggestions in Bing** **24-layer BERT**	**……**

Model operationalization with ONNX



Demo

[PyTorch BERT acceleration with ONNX Runtime]

1. Load Pretrained Bert model

We begin by downloading the SQuAD data file and store them in the specified location.

```
import os

cache_dir = "./squad"
if not os.path.exists(cache_dir):
    os.makedirs(cache_dir)

predict_file_url = "https://rajpurkar.github.io/
predict_file = os.path.join(cache_dir, "dev-v1.1
if not os.path.exists(predict_file):
    import wget
    print("Start downloading predict file.")
    wget.download(predict_file_url, predict_file
    print("Predict file downloaded.")
```

Specify some model configuration variables and constant.

```
# For fine tuned large model, the model name is
ere we use bert-base for demo.
model_name_or_path = "bert-base-cased"
max_seq_length = 128
doc_stride = 128
max_query_length = 64

# Enable overwrite to export onnx model and dowr
enable_overwrite = True

# Total samples to inference. It shall be large
total_samples = 100
```

Start to load model from pretrained. This step could take a f

```
# The following code is adapted from HuggingFace
# https://github.com/huggingface/transformers/bl

from transformers import (BertConfig, BertForQue

# Load pretrained model and tokenizer
config_class, model_class, tokenizer_class = (Be
config = config_class.from_pretrained(model_name
tokenizer = tokenizer_class.from_pretrained(mode
model = model_class.from_pretrained(model_name_c
                                    from_tf=Fals
                                    config=confi
                                    cache_dir=ca

# load some examples
from transformers.data.processors.squad import S

processor = SquadV1Processor()
examples = processor.get_dev_examples(None, file

from transformers import squad_convert_examples_
features, dataset = squad_convert_examples_to_fe
            examples=examples[:total_samples], #
            tokenizer=tokenizer,
            max_seq_length=max_seq_length,
            doc_stride=doc_stride,
            max_query_length=max_query_length,
            is_training=False,
            return_dataset='pt'
        )
```

2. Export the loaded moc

Once the model is loaded, we can export

```
output_dir = "./onnx"
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
export_model_path = os.path.join(

import torch
device = torch.device("cpu")

# Get the first example data to r
data = dataset[0]
inputs = {
    'input_ids':      data[0].to(
    'attention_mask': data[1].to(
    'token_type_ids': data[2].to(
}

# Set model to inference mode, wh
differently in
# inference and training mode.
model.eval()
model.to(device)

if enable_overwrite or not os.pat
    with torch.no_grad():
        symbolic_names = {0: 'bat
        torch.onnx.export(model,
                          args=tu
multiple inputs)
                          f=expor
be a file or file-like object)
                          opset_v
he model to
                          do_cons
folding for optimization
                          input_n


                          output_
                          dynamic_



        print("Model exported at
```

```
Model exported at  ./onnx\bert-ba
```

4. Inference ONNX Model with ONNX Runtime

OpenMP Environment Variable ¶

OpenMP environment variables are very important for CPU inference of Bert model. It has large performance impact on Bert model so you might need set it carefully according to [Performance Test Tool] result in later part of this notebook.

Setting environment variables shall be done before importing onnxruntime. Otherwise, they might not take effect.

```
import psutil

# You may change the settings in this cell according to Performance Test Tool result.
use_openmp = False

# ATTENTION: these environment variables must be set before importing onnxruntime.
if use_openmp:
    os.environ["OMP_NUM_THREADS"] = str(psutil.cpu_count(logical=True))
else:
    os.environ["OMP_NUM_THREADS"] = '1'

os.environ["OMP_WAIT_POLICY"] = 'ACTIVE'
```

Now we are ready to inference the model with ONNX Runtime. Here we can see that OnnxRuntime has better performance than PyTorch.

It is better to use standalone python script like [Performance Test tool] to get accurate performance results.

```
import onnxruntime
import numpy

# Print warning if user uses onnxruntime-gpu instead of onnxruntime package.
if 'CUDAExecutionProvider' in onnxruntime.get_available_providers():
    print("warning: onnxruntime-gpu is not built with OpenMP. You might try onnxruntime package to test CP
U inference.")

sess_options = onnxruntime.SessionOptions()

# Optional: store the optimized graph and view it using Netron to verify that model is fully optimized.
# Note that this will increase session creation time, so it is for debugging only.
sess_options.optimized_model_filepath = os.path.join(output_dir, "optimized_model_cpu.onnx")

if use_openmp:
    sess_options.intra_op_num_threads=1
else:
    sess_options.intra_op_num_threads=psutil.cpu_count(logical=True)

# Specify providers when you use onnxruntime-gpu for CPU inference.
session = onnxruntime.InferenceSession(export_model_path, sess_options, providers=['CPUExecutionProvide
r'])

latency = []
for i in range(total_samples):
    data = dataset[i]
    # Use contiguous array as input might improve performance.
    # You can check the results from performance test tool to see whether you need it.
    ort_inputs = {
        'input_ids':  numpy.ascontiguousarray(data[0].cpu().reshape(1, max_seq_length).numpy()),
        'input_mask': numpy.ascontiguousarray(data[1].cpu().reshape(1, max_seq_length).numpy()),
        'segment_ids': numpy.ascontiguousarray(data[2].cpu().reshape(1, max_seq_length).numpy())
    }
    start = time.time()
    ort_outputs = session.run(None, ort_inputs)
    latency.append(time.time() - start)
print("OnnxRuntime cpu Inference time = {} ms".format(format(sum(latency) * 1000 / len(latency), '.2f')))
```

ONNX Runtime Adoption

Up to 18x performance gains seen by Microsoft services

10+ platforms integrated with ONNX Runtime

Millions of devices running ONNX Runtime

Billions of requests handled in prod

SQL Server

ORACLE

Visual Studio Code

PowerApps

CERN

Power BI

AXIOMA

Microsoft Cognitive Services

Office 365

MarkLogic

Windows

Bing

QAS

skype

Deep Learning Inference Service (DLIS)

Azure Kinect DK

navitaire
an amadeus company

Thanks

ONNX ONNX Runtime BERT acceleration in ONNX Runtime