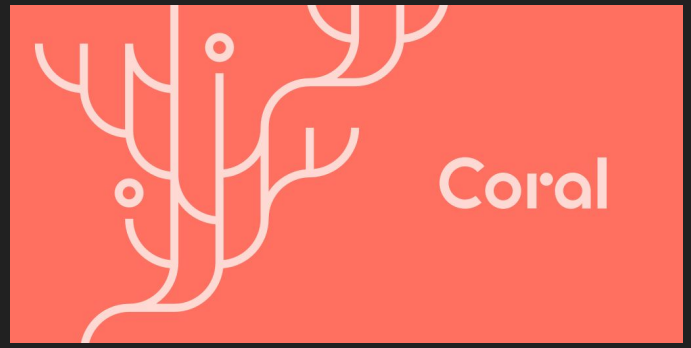
Optimizing Inference on Heterogeneous Platforms

Param Damle (psd9vgc), Pawan Jayakumar (pj8wfq)

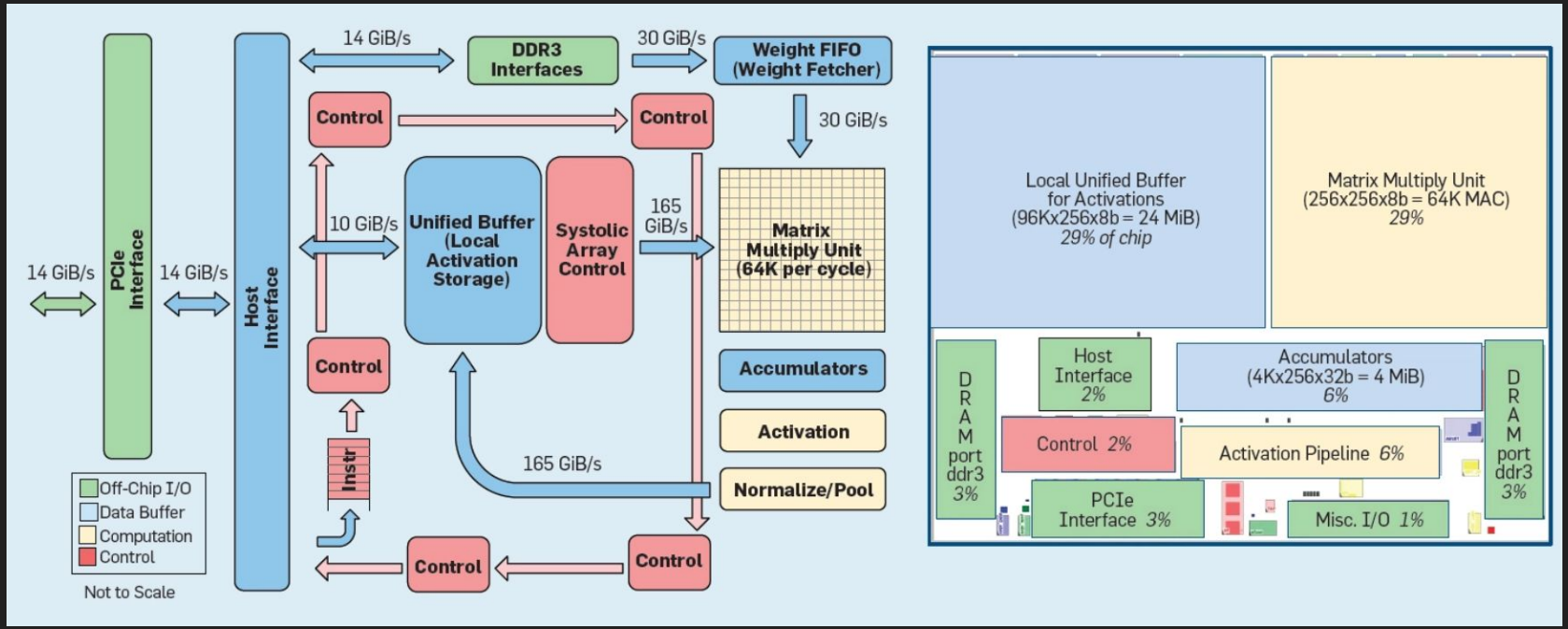
CS 6501: Hardware Accelerators

Background: Edge TPU

- Machine learning (ML) inference has become a significant workload in data centers
- Google's Edge TPU operates as a compact version of its regular TPU and is available commercially through Coral.AI



Background: Tensor Processing Unit (TPU)



Heterogeneous Platforms

- Due to the limitations of Moore’s law, accelerators for specific applications are becoming more prevalent
- How can servers and edge devices utilize multiple accelerators?
- We explore how a system can utilize both a TPU and GPU for ML inference

Prior Work in Heterogeneous Computing

- GPTPU: a CUDA-like framework for the Edge TPU.
  - generalize TPU beyond inference
- Simultaneous Heterogeneous Multi-Threading (SHMT)
  - data level parallelism
  - 1.95 × speedup and 51.0% energy reduction on DSP
- Shared Learning Among Distributed Edge Devices
  - model level parallelism

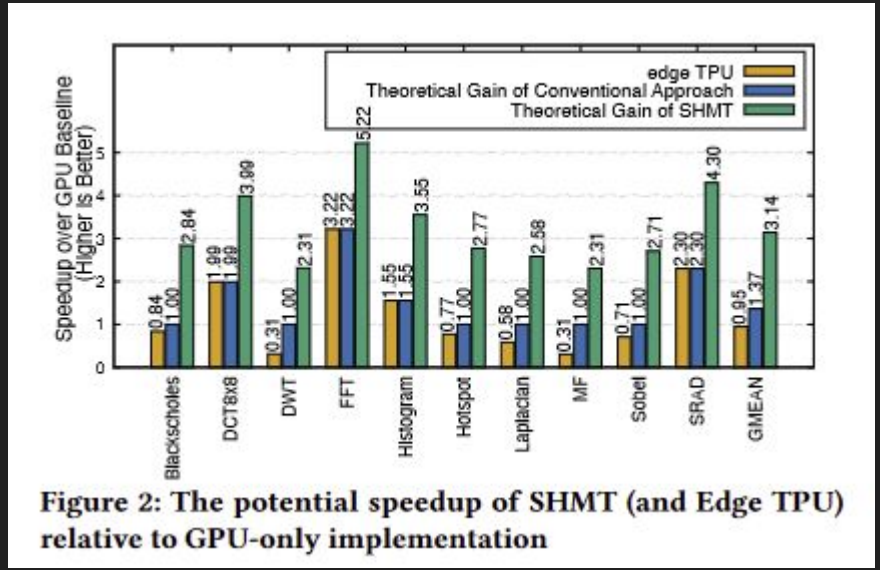
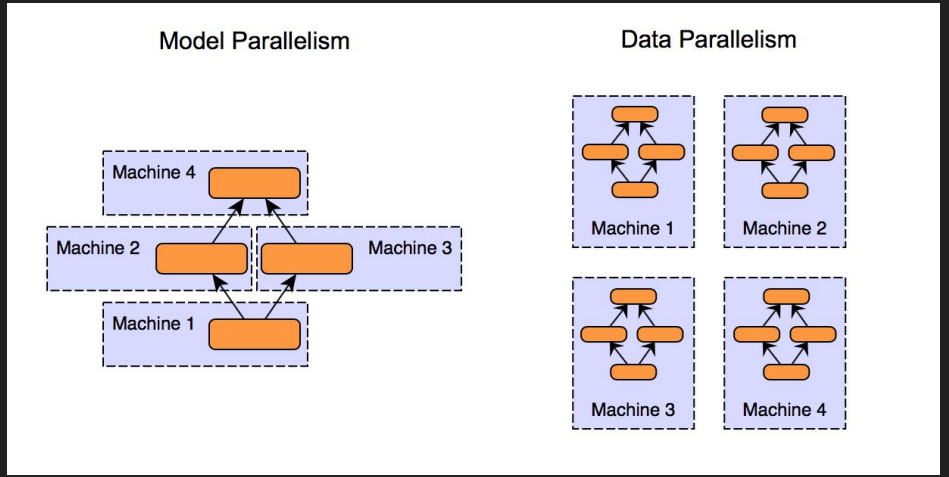


Figure 2: The potential speedup of SHMT (and Edge TPU) relative to GPU-only implementation

Research Interests

- Data Parallelism: split data among many devices
- Layer Parallelism: split model layers among many devices
- Model Alignment: which devices are best for which models?
- Quantization: how does data representation affect performance?
- Assess Performance Metrics: inference speed, accuracy, scalability

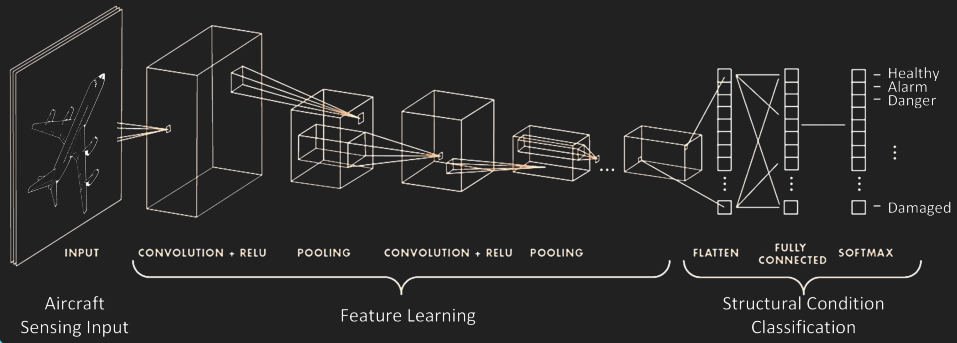


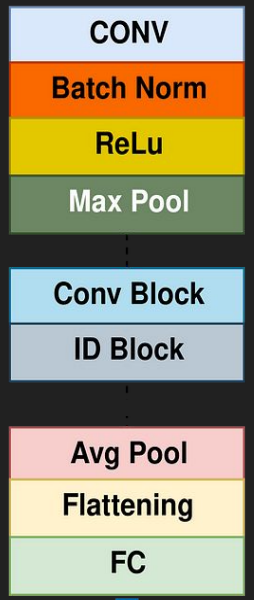
Methodology

Step 1: Create model in TensorFlow

```
model = Sequential()
model.add(keras.Input((input_dim,)))
model.add(Dense(output_dim, activation='linear', use_bias=False))#
model.compile()#optimizer='adam', loss='categorical_crossentropy')
model.summary()
# a = np.array(model.get_weights())        # save weights in a np
# model.set_weights(a + 1)                 # add 1 to all weights
# b = np.array(model.get_weights())        # save weights a secon
# print(b - a)                             # print changes in wei
```

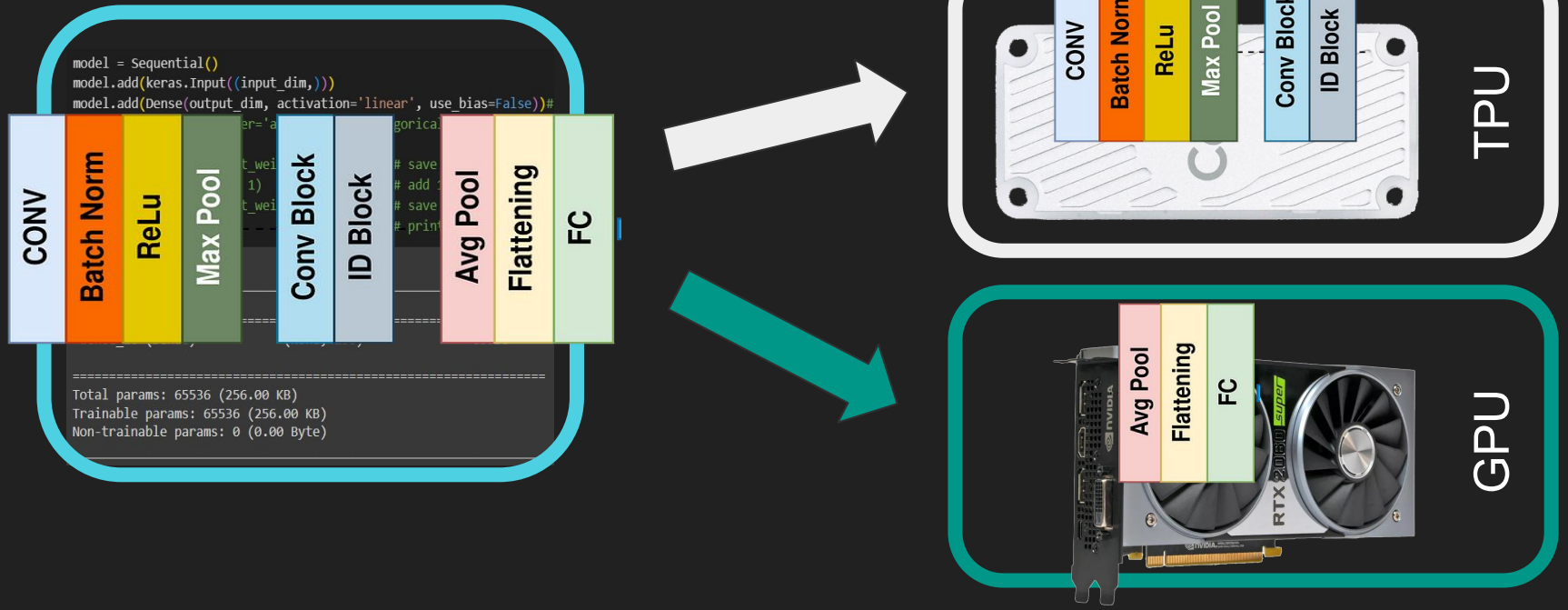
```
Model: "sequential_16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_16 (Dense)            (None, 256)               65536

=================================================================
Total params: 65536 (256.00 KB)
Trainable params: 65536 (256.00 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```





Methodology

Step 2: Shard model into GPU and TPU layers, if necessary



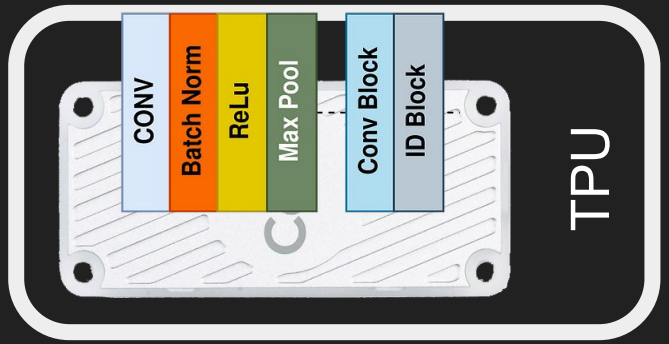
Methodology

Step 3a: Run GPU-optimized model using CUDA


```
inputshape = model.layers[0].input_shape[1:]
with tf.device('/gpu:0'):
# with tf.device('/cpu'):
    times = []
    for i in range(30):
        start = time.perf_counter()
        outputs = model.predict(np.random.rand(batch_size, *inputshape), verbose=0)
        end = time.perf_counter()
        times.append((end-start)*1000)
```

Methodology

Step 3b: Compile TPU model into TFLite and run inference using PyCoral wrapper



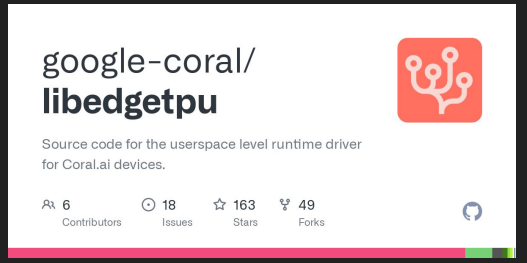

```
# # Load the TFLite model and allocate tensors.
interpreter = make_interpreter(tflite_path)  # edge tpu specific interpreter
interpreter.allocate_tensors()
```

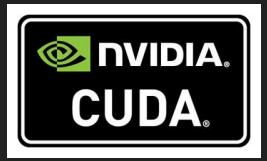
```
# # Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
output_scale, output_zero_point = output_details[0]['quantization']
input_scale, input_zero_point = input_details[0]["quantization"]
```

```
input_shape = tuple(input_details[0]['shape'])
# print(input_shape)
num_trials = 30
inference_times = []
for trial in range(num_trials):
    input_data = generate_input(input_shape)
    input_data = (input_data / input_scale) + input_zero_point
    interpreter.set_tensor(input_details[0]['index'], input_data.astype(np.uint8))
    start = time.perf_counter()
    interpreter.invoke()
    end = time.perf_counter()
    output_data = interpreter.get_tensor(output_details[0]['index'])
    output_data = output_scale * (output_data - output_zero_point)
    inference_time = (end - start) * 1000
    inference_times.append(inference_time)
```

Roadblocks

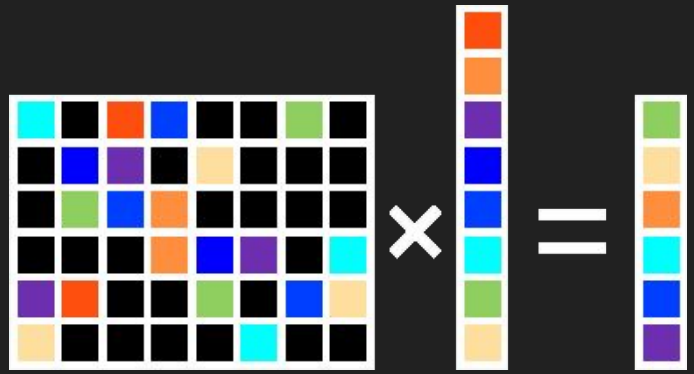
- Setting up Edge TPU on Windows or WSL
  - Difficulties installing Edge TPU runtime
  - USB incompatibility
  - Coral TFLite incompatibility with TensorFlow TFLite
- Running models on GPU
  - Windows GPU integration required old TensorFlow < 2.11
- Splitting TensorFlow models
  - TF was not designed to split layer-wise and have separably runnable models
- Quantization is fixed to uint8 on Edge TPU, FP16 on Nvidia RTX 2060





Results: Matrix Multiplication

We started by benchmarking regular matrix-vector multiplication, the foundational operation both GPU and TPU were designed to optimize

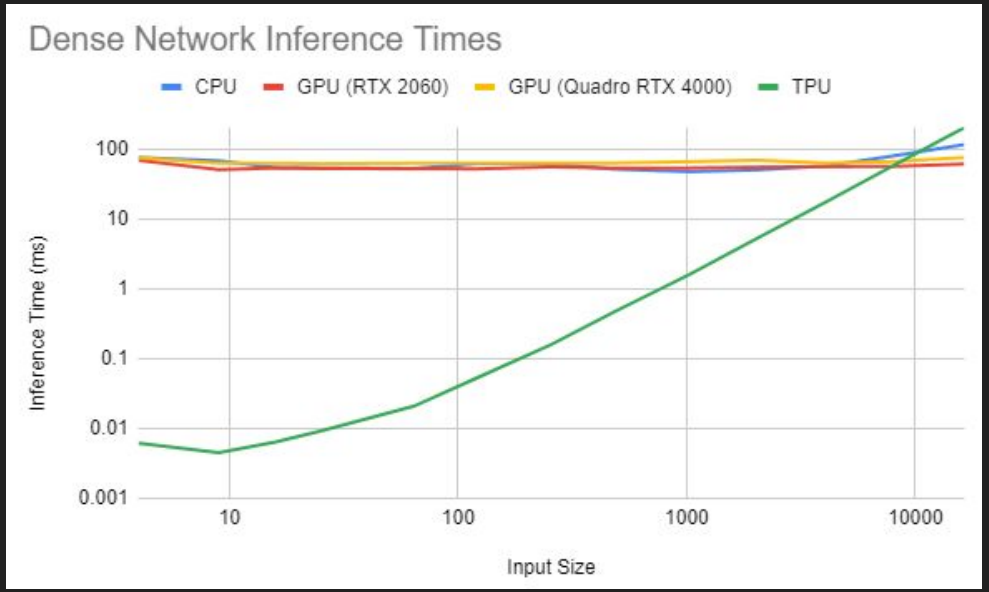


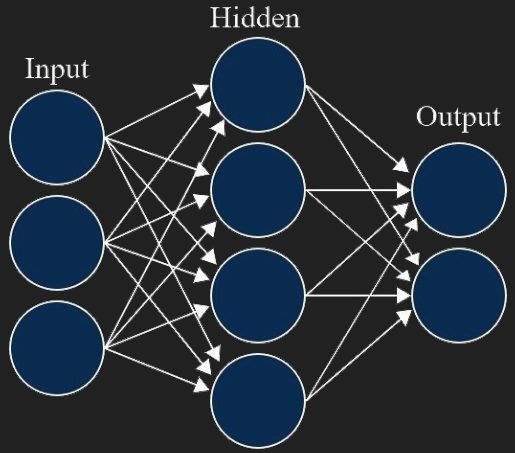
Matrix Side Length	64	128	256	512	1024
Edge TPU (with memory loading)	**0.1518**	**0.3469**	0.9515	3.5786	14.0721
NVIDIA GTX 1650 (with memory loading)	0.2009	0.3865	**0.4294**	**0.6712**	**1.3720**
Edge TPU (inference only)	0.0586	0.2931	0.9932	3.4194	14.0038
NVIDIA GTX 1650 (inference only)	0.0624	0.1252	0.0718	0.0930	**0.1106**
Intel Xeon @ 2.2GHz	**0.0412**	**0.0687**	**0.0518**	**0.0661**	0.1555

Table 1. Table of inferencing time (in milliseconds). All results shown are the average of 100 trials

Results: Scaling Dense Neural Net

The basic machine learning workload is a feed-forward neural network:



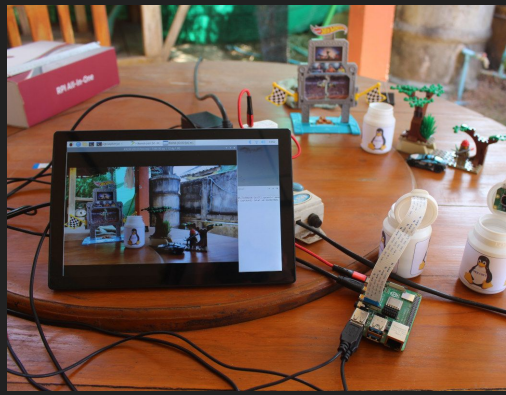


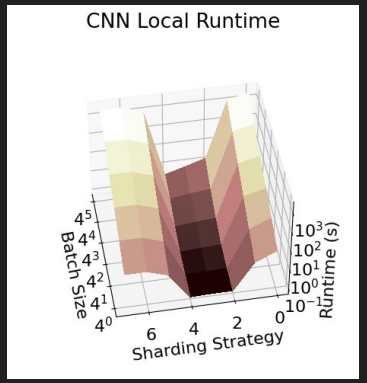
Although scaling less efficiently,

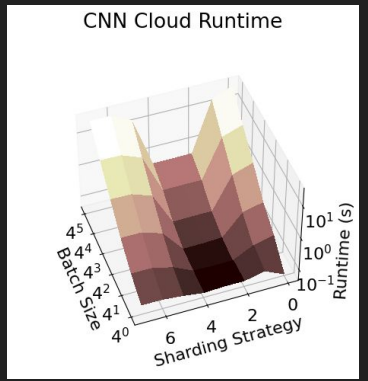
TPU outperforms and single-sample inference for <10,000 input size (100x100 black and white image)

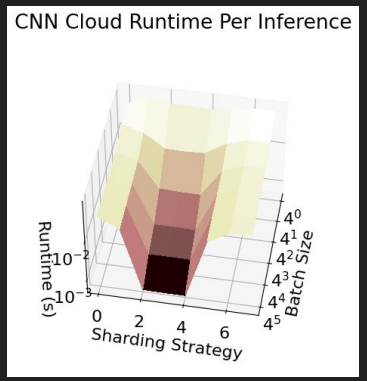
Results: Sharding Convolutional Neural Net

Edge devices (e.g. camera systems) are often used for computer vision tasks



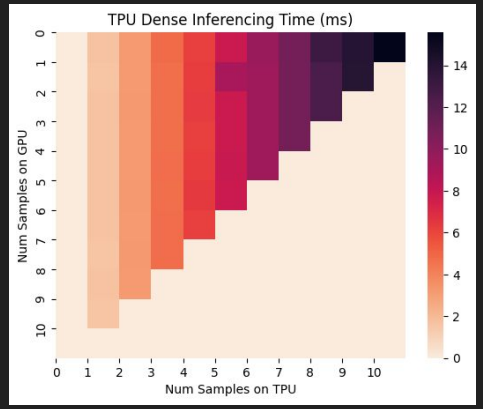


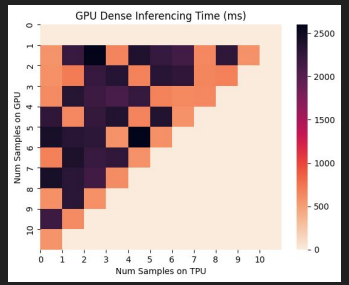


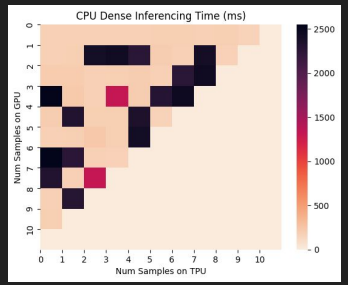


Results: Data Parallelism in small, dense network

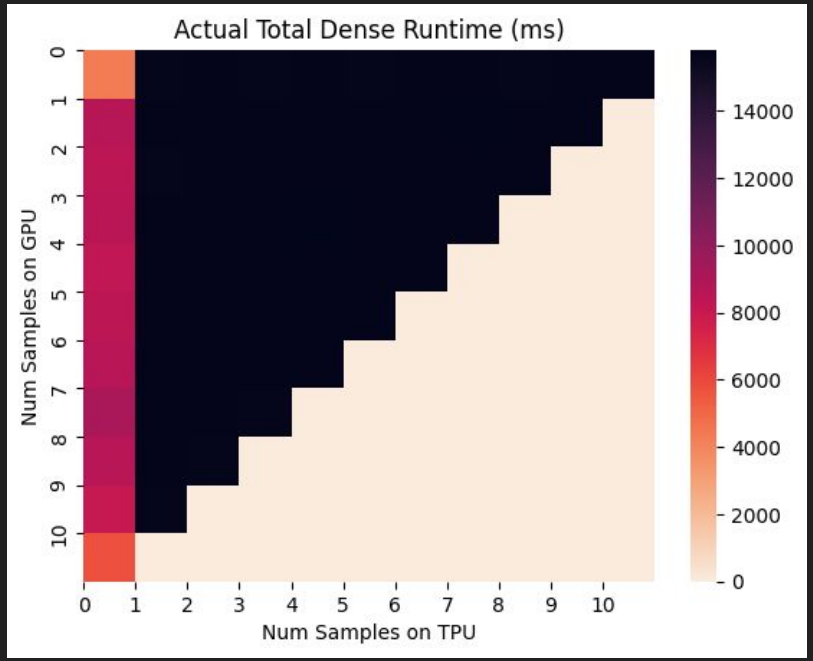
Edge TPU's dominance at small scale balanced with its serial and USB limitations





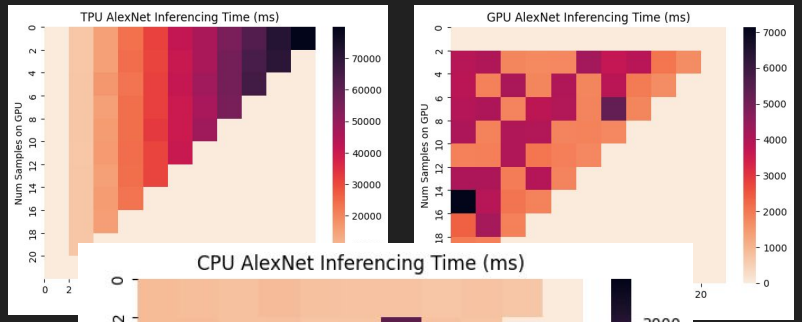


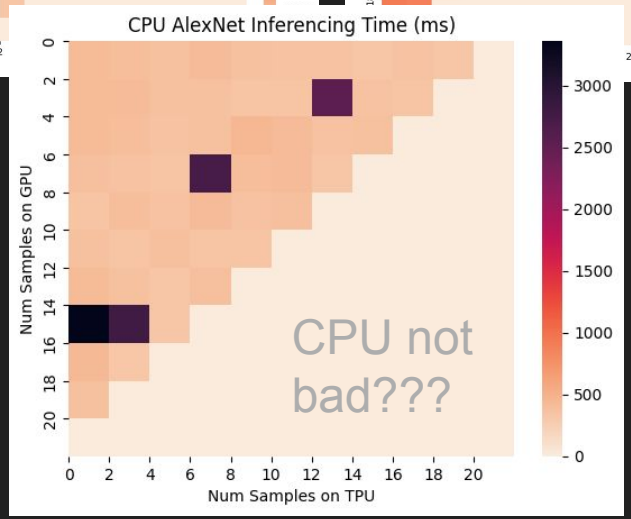
TPU introduces some fixed overhead?



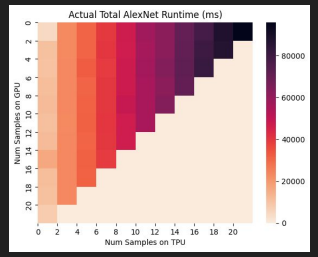
Results: Data Parallelism in large AlexNet CNN

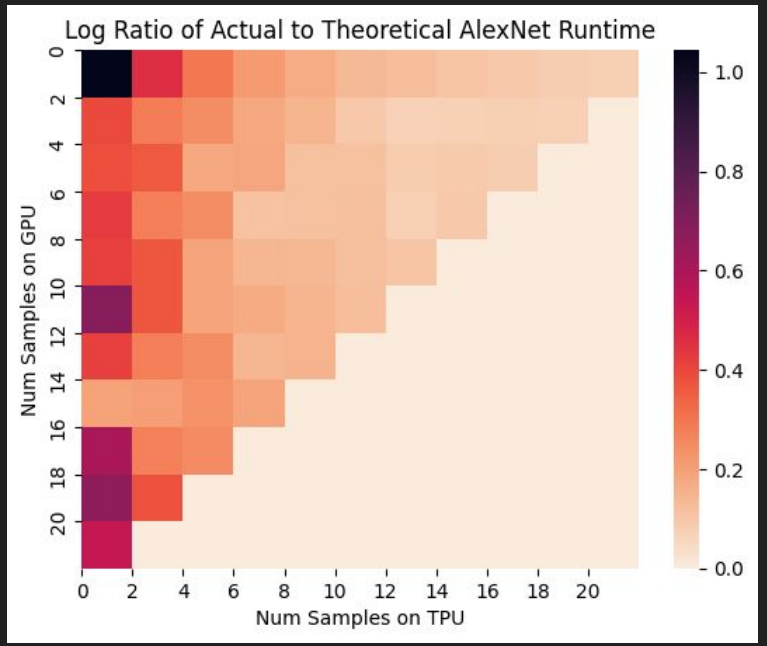
Edge TPU delivers slow but reliable performance





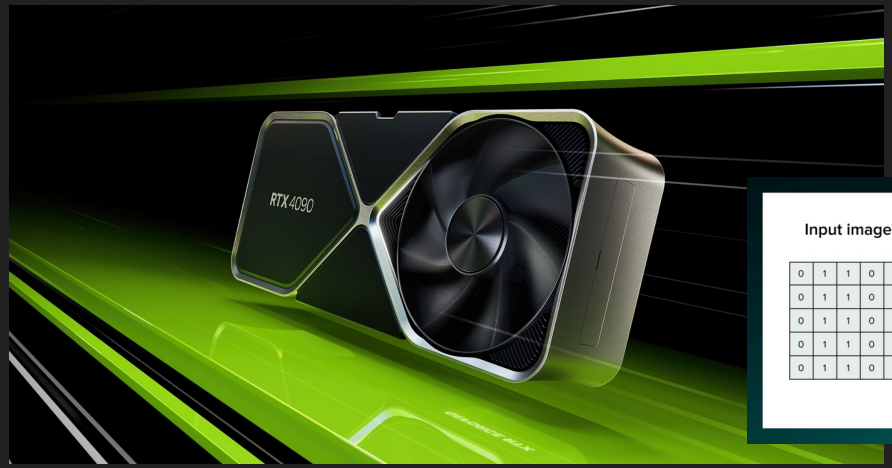
(domination)

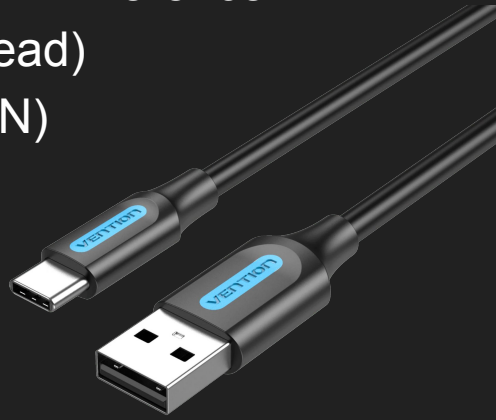


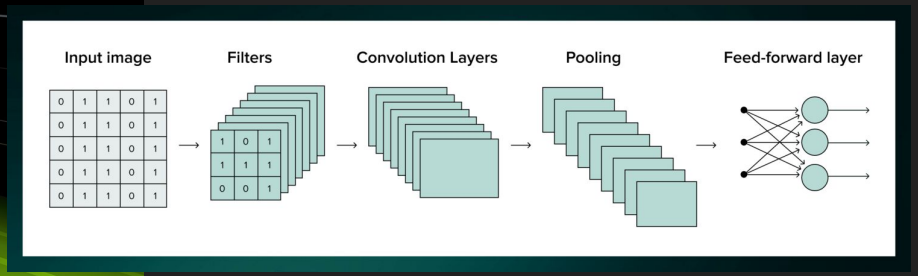


Takeaways

- Edge TPU is best for single-sample, small-scale, simple DNN inference
- GPUs performed better at large batches (amortizing overhead)
- GPUs better handled more complex architectures (e.g. CNN)
- USB connection was major bottleneck







Future Work

- Lack of untethered edge-scale processing units
  - No easily accessible mobile phone CPUs and GPUs to connect and test
- Lack of ready to use toolkits to benchmark across different devices
  - every device requires its own software stack which makes it difficult to compare
  - creating toolkits from scratch is difficult
- Generalization or modernization of TPU
  - ML workloads have evolved considerably from simple neural networks