Cloud TPU provides high performance at low cost. You can enhance Cloud TPU performance further by adjusting Cloud TPU configuration parameters for your application and by identifying and resolving any bottlenecks that are limiting performance.

This guide helps you maximize Cloud TPU performance by showing you how to:

- Improve XLA compiler efficiencies (#xla-efficiencies).
- Identify and use a set of tuneable TensorFlow functions (#tf-fcts).

In addition, the following resources describe how to:

- Tune the input pipeline (https://www.tensorflow.org/versions/master/performance/datasets_performance).
- Use Cloud TPU tools (/tpu/docs/cloud-tpu-tools) to identify performance bottlenecks.
- Improve overall performance to converge faster (https://www.youtube.com/watch?v=SxOsJPaxHME).

These pages only provide a set of guidelines. It might be necessary to deviate from their advice for any number of rea
ample, fundamental model architecture requirements). The guidelines will change over time as improvements are ma
oud TPU software stack.

XLA is a compiler for machine learning that can produce binaries for TPUs, CPUs, GPUs and other platforms. It is part of the standard TensorFlow code base. TensorFlow models for Cloud TPU are translated to an XLA graph, which XLA then compiles to a TPU executable. More details about how XLA and TensorFlow interact are included in the XLA overview (https://www.tensorflow.org/performance/xla/).

The Cloud TPU hardware is different from CPUs and GPUs. At a high level, CPUs can be characterized as having a low number of high performing threads. GPUs can be characterized as having a very high number of low performing threads. A Cloud TPU, with its 128 x 128 matrix unit, can be thought of as either a single, very powerful thread, which can perform 16K ops per cycle, or 128 x 128 tiny, simple threads that are connected in pipeline fashion. Correspondingly, when addressing memory,

multiples of 8 (floats) are desirable, as well as multiples of 128 for operations targeting the matrix unit.

Arrays in the Cloud TPU are tiled. This entails padding one of the dimensions to a multiple of 8, and a different dimension to a multiple of 128. XLA performs data layout transformations and data is arranged in memory such that the hardware can efficiently process the data. These transformations are driven by heuristics. While they are beneficial in most cases, there is always potential for the compiler to do the wrong thing. To achieve the highest performance, it can be beneficial to experiment with different model configurations.

One consequence of using a tiled memory scheme is that efficient memory utilization is dependent on the amount of memory wasted on padding overhead. To use the Cloud TPU in the most efficient way, use dimension sizes that minimize the overhead of tiling.

For example, a convolution result has (1) batch, (2) output feature, and (3) output spatial dimensions. One of either the batch or output feature dimensions will be padded to a multiple of 8 while the other will be padded to a multiple of 128. The output spatial dimensions will not be padded.

In general, for an operation with spatial dimensions (`tf.nn.pool`, `tf.conv2d`, etc.), the spatial dimensions are never padded.

Batch and feature dimensions are subject to padding, so be careful when determining the batch and feature sizes. To make best use of the 128 x 128 matrix unit, strive for reasonably large values (>=128) of batches or features, preferably both.

In most cases, a batch size of 128 is sufficient to keep the Cloud TPU matrix unit fully occupied. Running with larger batch sizes also works well on the Cloud TPU but using a batch size that is a multiple of 128 is recommended. If your model cannot work in such a configuration, try to use a batch size that is a multiple of 8, to minimize the impact of padding.

Similarly, feature dimensions are also mapped to a dimension which is padded to either 8 or 128, depending on decisions made by the XLA layout algorithm. This means that the feature dimension wastes the least amount of space at multiples of either 8 or 128.

Fusion is a general technique that the XLA compiler uses to optimize programs. A fused operation is the combination of multiple constituent operations that are to be executed in combination.

For example, consider the following series of operations:

This code is roughly equivalent to the following pseudo code:

With fusion, the array accesses happen at the same time:

In this example, the number of memory round trips is reduced and XLA didn't need to allocate any space for 'tmp'.

Fusion is a critical optimization and benefits the Cloud TPU in several ways:

- It reduces memory transfers by removing the need to store intermediate results in main memory, which is slow.
- It allows greater utilization of hardware units which would otherwise be unutilized.
- It can reduce the memory utilization of a model as fewer buffers need to be live at the same time.

Broadcasting implicitly occurs when two tensors with different, but compatible, shapes are combined.

For example, `tf.add(vector, matrix)` requires the vector to be broadcasted to the shape of the matrix. The result of the operation has the same shape as the matrix. For more details, see the guide to broadcasting arrays (https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html).

While broadcasts can often be fused with their consumers, forcing a broadcast to be materialized results in poor performance and increased memory pressure.

In the following example, the broadcast implicit in the addition of a vector and matrix cannot be fused with the argmax resulting in a materialized broadcast:

See the full list of TensorFlow operations (https://cloud.google.com/tpu/docs/tensorflow-ops) available on Cloud TPU.

- Transposing the result of either of the operands is effectively free.

- Note that `tf.matmul` supports fusing into its input and output. This means that activation functions or biases applied directly to the output of `tf.matmul` have low overhead.

- For the activations, the batch and feature dimensions are padded to a multiple of either 8 or 128.
    - First XLA tracks the most common size of batch dimensions for convolutions in the module. This helps distinguish between forward convolutions, activation gradient convolutions, and kernel gradient convolutions.
    - If the most common batch size is greater than or equal to 64:
        - Batch is padded to a multiple of 128 and feature padded to a multiple of 8 for forwards and backwards convolutions.
        - Batch is padded to a multiple of 8 and feature padded to a multiple of 128 for gradient update convolutions.
    - If the most common batch size is less than 64:
        - Batch is padded to a multiple of 8 and feature padded to a multiple of 128 for forwards and backwards convolutions.
        - Batch is padded to a multiple of 128 and feature padded to a multiple of 8 for gradient update convolutions.
        - Transposing the activations right before sending it to a convolution is free if the transpose only swaps the input feature and batch dimensions.
- For the kernel, the input feature and output feature dimensions are padded to a multiple of either 8 or 128. The exact determination is influenced by the producers and other consumers of the kernel.
    - Transposing a kernel right before sending it to a convolution is free if the transpose only swaps the input and output feature dimensions.
- For the result, the batch and feature dimensions are padded to a multiple of either 8 or 128.
    - Transposing the result of a convolution is free if the transpose only swaps the batch and output feature dimensions.
- Note that `tf.nn.conv_n_d` supports fusing into its result, the activations and/or the kernel. This means that activation functions or biases applied directly to the output have low overhead.

- The padding rules apply: spatial dimensions are more major than batch and feature. Each of batch and feature may be padded to a multiple of either 8 or 128.
- Typically, the layout of a pool operation matches the convolutions that flow in or out of it.
- The gradient calculation for `tf.nn.max_pool` may be slower than their `tf.nn.avg_pool` equivalent. Consider switching from max-pooling to average-pooling when possible.

- Avoid unnecessary slices and concatenations. Slices and concatenations in a dimension that has been padded is considerably more expensive.
    - Data movement is minimized if the slice dimension has no padding overhead.

- Transposing any of the operands of a `tf.matmul` or its result are free.
- Transposing the activations of a `tf.conv_n_d` is free if it swaps the batch and input feature dimensions.
- Transposing the kernel of a `tf.conv_n_d` is free if it swaps the input and output feature dimensions.
- Transposing the result of a `tf.conv_n_d` is free if it swaps the batch and output feature dimensions.

- These are costly because they involve moving data from padded to unpadded dimensions and vice-versa.

- Reshaping may be costly on Cloud TPU when moving around data in a padded dimension.
- It can be beneficial to reshape data to R1 on the host and reshape it back to some higher dimension shape on the device if there is substantial padding. This can make transfers

between host and device more efficient.

    - It can also help with peak memory utilization because the packed parameter can be unpacked on-demand.

- Pseudo random-number generation for uniform or Bernoulli distributions is very fast.
- Normal distributions are slightly more expensive than uniform or Bernoulli distributions.
- Pseudo random-number generation for Categorical/Multinomial distributions is considerably more expensive.

- Multiple reductions with the same input and output shape can be performed in parallel via fusion.
    - Try to rewrite sequential chains of reductions into parallel ones, if possible.
- Reductions support fusing elementwise operations into their input but not their output. When possible, rewrite expressions to promote fusion. For example:

  Into:

- The Cloud TPU compiler can efficiently lower TensorFlow's fused variants of batch normalization. Using them can be considerably more efficient than the alternative.

- Prefer `tf.nn.fused_batch_norm` over `tf.nn.batch_normalization`.
- For `tf.layers.batch_normalization`, set the "fused" argument to true.

- These are not fully optimized yet and may have worse performance than an equivalent, normal convolution.