Edge TPU Compiler

The Edge TPU Compiler (`edgetpu_compiler`) is a command line tool that compiles a TensorFlow Lite model (`.tflite` file) into a file that's compatible with the Edge TPU. This page describes how to use the compiler and a bit about how it works.

Before using the compiler, be sure you have a model that's compatible with the Edge TPU. For compatibility details, read TensorFlow models on the Edge TPU.

System requirements

The Edge TPU Compiler can be run on any modern Debian-based Linux system. Specifically, you must have the following:

- 64-bit version of Debian 6.0 or higher, or any derivative thereof (such as Ubuntu 10.0+)
- x86-64 system architecture

If your system does not meet these requirements, try our web-based compiler using Google Colab.

> **Note:** The Edge TPU Compiler is no longer available for ARM64 systems (such as the Coral Dev Board), beginning with version 2.1. We recommend compiling your model on a more powerful desktop.

Download

You can install the compiler on your Linux system with the following commands:

```
curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -

echo "deb https://packages.cloud.google.com/apt coral-edgetpu-stable main" | sudo tee /etc/

sudo apt-get update

sudo apt-get install edgetpu-compiler
```

Usage

```
edgetpu_compiler [options] model...
```

The compiler accepts the file path to one or more TensorFlow Lite models (the `model` argument), plus any options. If you pass multiple models (each separated with a space), they are co-compiled such that they can share the Edge TPU's RAM for parameter data caching (read below about **parameter data caching**).

The filename for each compiled model is *`input_filename`*`_edgetpu.tflite`, and it is saved to the current directory, unless you specify otherwise with the `-out_dir` option.

Table 1. Available compiler options

Option	Description
`-h, --help`	Print the command line help and exit.
`-i, --intermediate_tensors` *names*	Specify the tensors you want as outputs from the Edge TPU custom op. The argument is a comma-separated list of tensor names. All operators following these tensors will not be compiled for the Edge TPU but are still in the `.tflite` file and will run on the CPU. The final output from the model remains the same. If you have multiple input models (for co-compilation), then separate the list of tensors for each model with a colon, and pass them in the order corresponding to the input models.
`-m, --min_runtime_version` *num*	Specify the lowest Edge TPU runtime version you want the model to be compatible with. For example, if the device where you plan to execute your model has version 10 of the Edge TPU runtime (and you can't update the runtime version), then you should set this to 10 to ensure your model will be compatible. (Models are always forward-compatible with newer Edge TPU runtimes; a model compiled for version 10 runtime is compatible with version 12.) The default value depends on your version of the compiler; check the `--help` output. See below for more detail about the **compiler and runtime versions**.
`-n, --num_segments` *num*	Compile the model into *num* segments. Each segment can run on a separate Edge TPU, thus increasing overall model throughput. For usage details, see **Segmenting a model for pipelining**. This cannot be used when passing multiple models or combined with `intermediate_tensors`.
`-o, --out_dir` *dir*	Output the compiled model and log files to directory *dir*. Default is the current directory.
`-s, --show_operations`	Print the log showing operations that mapped to the Edge TPU. The same information is always written in a `.log` file with the same name and location as the compiled model.

Option	Description
`-d, --search_delegate`	Enable repeated search for a new compilation stopping point earlier in the graph, to avoid rare compiler failures when it encounters an unsupported operation. This flag might be useful if the compiler fails to create an Edge TPU delegate even though early layers of the model are compatible. When enabled, if the compiler fails for any reason, it automatically re-attempts to compile but will stop compilation before reaching the operation that failed. If it fails again, it repeats, again stepping backward and stopping before the failed op, repeating until it reaches the graph's very first op. The step distance for each recursive attempt is 1 op, by default. You can modify the step size with `--delevate_search_step`. This flag cannot be used when co-compiling multiple models. Added in v16.
`-k, --delegate_search_step` *num*	Specify a step size (the number of ops to move backward) when using `--search_delegate`, which uses a step size of 1 by default. This can be any number 1 or higher, which can help you find a successful delegate faster. Added in v16.
`-t, --timeout_sec` *num*	Specify the compiler timeout in seconds. The default is 180.
`-v, --version`	Print the compiler version and exit.

Parameter data caching

The Edge TPU has roughly 8 MB of SRAM that can cache the model's parameter data. However, a small amount of the RAM is first reserved for the model's inference executable, so the parameter data uses whatever space remains after that. Naturally, saving the parameter data on the Edge TPU RAM enables faster inferencing speed compared to fetching the parameter data from external memory.

This Edge TPU "cache" is not actually traditional cache—it's compiler-allocated scratchpad memory. The Edge TPU Compiler adds a small executable inside the model that writes a specific amount of the model's parameter data to the Edge TPU RAM (if available) before running an inference.

When you compile models individually, the compiler gives each model a unique "caching token" (a 64-bit number). Then when you execute a model, the Edge TPU runtime compares that caching token to the token of the data that's currently cached. If the tokens match, the runtime uses that cached data. If they don't match, it wipes the cache and writes the new model's data instead. (When models are compiled individually, only one model at a time can cache its data.) This process is illustrated in figure 1.

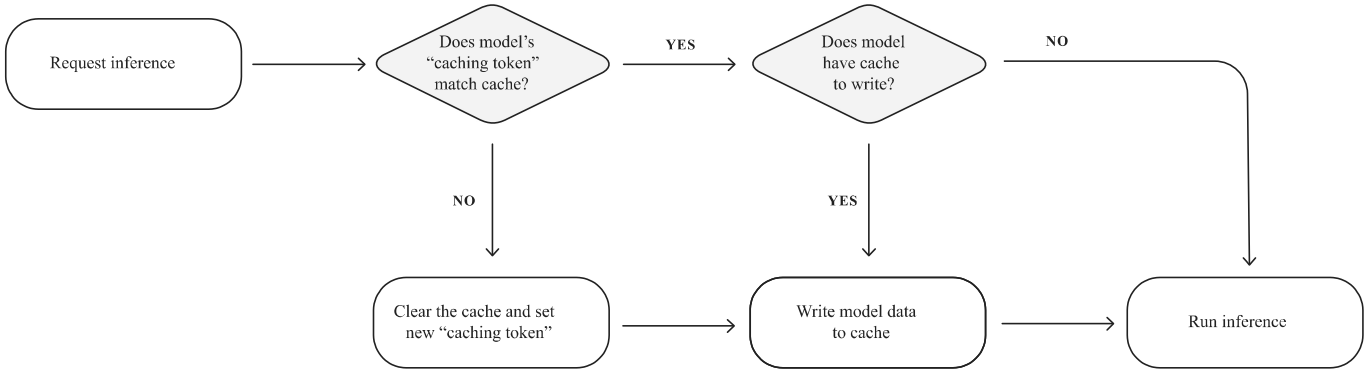

Figure 1. Flowchart showing how the Edge TPU runtime manages model cache in the Edge TPU RAM

Notice that the system clears the cache and writes model data to cache only when necessary, and doing so delays the inference. So the first time your model runs is always slower. Any later inferences are faster because they use the cache that's already written. But if your application constantly switches between multiple models, this cache swapping adds significant overhead to your application's overall performance. That's where the co-compilation feature comes in...

Co-compiling multiple models

To speed up performance when you continuously run multiple models on the same Edge TPU, the compiler supports co-compilation. Essentially, co-compiling your models allows multiple models to share the Edge TPU RAM to cache their parameter data together, eliminating the need to clear the cache each time you run a different model.

When you pass multiple models to the compiler, each compiled model is assigned the same caching token. So when you run any co-compiled model for the first time, it can write its data to the cache without clearing it first. Look again at figure 1: When you co-compile multiple models, the first decision node ("Does the model's caching token match cache?") becomes "Yes" for each model, and the second node is "Yes" only the first time that each model runs.

But beware that the amount of RAM allocated to each model is fixed at compile-time, and it's prioritized based on the order the models appear in the compiler command. For example, consider if you co-compile two models as shown here:

```
edgetpu_compiler model_A.tflite model_B.tflite
```

In this case, cache space is first allocated to model A's data (as much as can fit). If space remains after that, cache is given to model B's data. If some of the model data cannot fit into the Edge TPU RAM, then it must instead be fetched from the external memory at run time.

If you co-compile several models, it's possible some models don't get any cache, so they must load all data from external memory. Yes, that's slower than using the cache, but if you're running the models in quick succession, this could still be faster than swapping the cache every time you run a different model.

> **Note:** Parameter data is allocated to cache one layer at a time—either all parameter data from a given layer fits into cache and is written there, or that layer's data is too big to fit and all data for that layer must be fetched from external memory.

Performance considerations

It's important to remember that the cache allocated to each model is not traditional cache, but compiler-allocated scratchpad memory.

The Edge TPU Compiler knows the size of the Edge TPU RAM, and it knows how much memory is needed by each model's executable and parameter data. So the compiler assigns a fixed amount of cache space for each model's parameter data at compile-time. The `edgetpu_compiler` command prints this information for each model given. For example, here's a snippet of the compiler output for one:

```
On-chip memory available for caching model parameters: 6.91MiB
On-chip memory used for caching model parameters: 4.21MiB
Off-chip memory used for streaming uncached model parameters: 0.00B
```

In this case, the model's parameter data all fits into the Edge TPU RAM: the amount shown for "Off-chip memory used" is zero.

However, if you co-compile two models, then this first model uses 4.21 MiB of the available 6.91 MiB of RAM, leaving only 2.7 MiB for the second model. If that's not enough space for all parameter data, then the rest must be fetched from the external memory. In this case, the compiler prints information for the second model such as this:

```
On-chip memory available for caching model parameters: 2.7MiB
On-chip memory used for caching model parameters: 2.49MiB
Off-chip memory used for streaming uncached model parameters: 4.25MiB
```

Notice the amount of "Off-chip memory used" for this second model is 4.25 MiB. This scenario is roughly illustrated in figure 2.

> **Note:** The "On-chip memory available" that appears for the first co-compiled model is what's left after setting aside memory required by the model executables. If you co-compile multiple models, the space set aside for executables is shared between all models (unlike space for parameter data). That is, the amount given for the executables is only the amount of space required by the largest executable (not the sum of all executables).

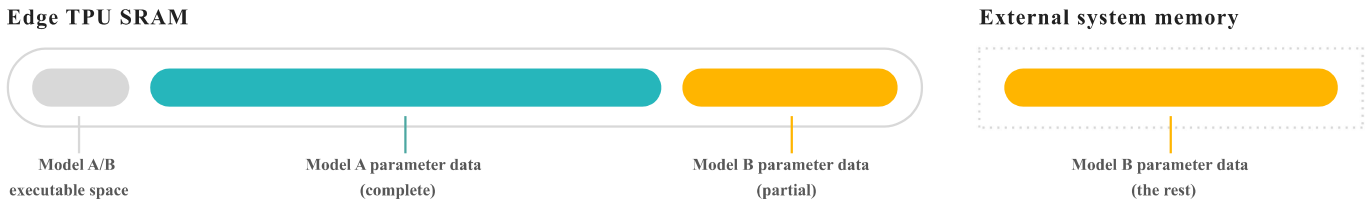

Figure 2. Two co-compiled models that cannot both fit all parameter data on the Edge TPU RAM

Even if your application then runs *only* this second model ("model B"), it will always store only a portion of its data on the Edge TPU RAM, because that's the amount determined to be available when you co-compiled it with another model ("model A").

The main benefit of this static design is that your performance is deterministic when your models are co-compiled, and time is not spent frequently rewriting the RAM. And, of course, if your models do fit all parameter data into the Edge TPU RAM, then you achieve maximum performance gains by never reading from external memory and never rewriting the Edge TPU RAM.

When deciding whether to use co-compilation, you should run the compiler with all your models to see whether they can fit all parameter data into the Edge TPU RAM (read the compiler output). If they can't all fit, consider how frequently each model is used. Perhaps pass the most-often-used model to the compiler first, so it can cache all its parameter data. If they can't all fit and they switch rarely, then perhaps co-compilation is not beneficial because the time spent reading from external memory is more costly than periodically rewriting the Edge TPU RAM. To decide what works best, you might need to test different compilation options.

> **Tip:** If the data for multiple models doesn't all fit in the cache, try passing the models to `edgetpu_compiler` in a different order. As mentioned above, the data is allocated one layer at a time. Thus, you might find an order that fits more total data in the cache because it allows in more of the smaller layers.

> **Caution:** You must be careful if you use co-compilation in combination with **multiple Edge TPUs**—if you co-compile your models but they actually run on separate Edge TPUs, your models might needlessly store parameter data on the external memory. So you should be sure that any co-compiled models actually run on the same Edge TPU.

Segmenting a model for pipelining

If your model is unable to fit into the **Edge TPU cache** or the overall throughput for your model is a bottleneck in your program, you can probably improve your performance by segmenting your model into separate subgraphs that run in a pipeline on separate Edge TPUs.

To segment your model, just specify the `num_segments` argument when you compile the model. For example, if you want a pipeline using four Edge TPUs:

```
edgetpu_compiler --num_segments=4 model.tflite
```

The compiler outputs each segment as a separate `.tflite` file with an enumerated filename. You can then run inference using the `PipelinedModelRunner` API, as described in **Pipeline a model with multiple Edge TPUs**.

However, you should carefully consider what number of segments will achieve your performance goals. This depends on the size of your model and whether you're trying to fit a large model entirely into the Edge TPU cache (to reduce latency) or you're trying to increase your model's throughput:

- If you just need to fit your model into the Edge TPU cache, then you can incrementally increase the number of segments until the compiler prints "Off-chip memory used" is 0.00B for all segments.

- If you want to increase the model throughput, then finding the ideal number of segments might be a little trickier. That's because although the Edge TPU Compiler divides your model so that each segment has roughly the same amount of parameter data, each segment can still have different latencies. For example, one layer might receive much larger input tensors than others, and that added processing can create a bottleneck in your pipeline. So you might improve throughput further by simply adding an extra segment. Based on our experiments, we found the following formula creates a well-distributed pipeline:

  ```
  num_segments = [Model size] MB / 6 MB
  ```

  Then round up to a whole number. For example, if your model is 20 MB, the result is 3.3, so you should try 4 segments.

  If the total latency is still too high, then it might be because one segment is still much slower than the rest and causing a bottleneck. To address the differing segment latencies, you should **try our profiling-based partitioner**, which profiles the latency for each segment and then re-segments the model to balance the latency across all segments.

> **Note:** If you want complete control of where each segment is cut, you can instead manually cut your model into separate `.tflite` files using the **TensorFlow `toco_convert` tool**. Be sure you name each `.tflite` segment with a number corresponding to the order it belongs in the pipeline. Also beware that `toco_convert` is only compatible with models using uint8 quantized parameters, so it's not compatible with post-training quantized models (which uses int8).

Using the profiling-based partitioner

The Edge TPU Compiler's segmentation strategy (via the `num_segments` option) uses a heuristic that tries to evenly distribute the parameter data between segments—it **does not** care about the latency in each segment. So it may leave bottlenecks in your pipeline, especially when trying to segment an SSD model or another type of model that has large CPU-bound ops or branches in the graph. For these situations where it's important that each segment of your pipeline perform at reasonably-similar latencies, you should compile your model using the `partition_with_profiling` tool.

The `partition_with_profiling` tool is a wrapper around the Edge TPU Compiler that profiles each segment on an Edge TPU and iteratively re-compiles the model using different segment lengths until it finds a partition ratio where the difference between fastest and slowest segment is below a given threshold.

To use this tool, you must first compile the `partition_with_profiling` tool, which you can do with Docker as shown in the **partitioner's GitHub readme**. Then run the profiling partitioner by passing it the path to the Edge TPU Compiler, your model, and the number of segments to create:

> **Notice:** Your system must have access to the same number of Edge TPUs as given to the `num_segments` argument.

```
./partition_with_profiling \
  --edgetpu_compiler_binary <PATH_TO_COMPILER> \
  --model_path <PATH_TO_MODEL> \
  --output_dir <OUT_DIR> \
  --num_segments <NUM_SEGMENTS>
```

This will take longer than a typical model compilation, because it may need to re-partition the model several times and profile each one before finding an optimal partition.

> **Note:** To accurately profile your pipeline segments, the Edge TPUs used by the profiling-based partitioner should use the same interface (PCIe or USB) as those in your production system, and the host CPU should also match. If that's not possible, that's okay, but beware there may be a difference between the throughput measured by the profiling-based partitioner and the throughput on your production system.

Table 2. Options for `partition_with_profiling` tool

Option	Description
`--edgetpu_compiler_binary` *`file_path`*	Path to the edgetpu compiler binary. **Required.**
`--model_path` *`file_path`*	Path to the model to be partitioned. **Required.**
`--num_segments` *`num`*	Number of output segment models. **Required.**
`--output_dir` *`dir`*	Output directory for all compiled segments. Default is `/tmp/models/`.
`--device_type` *`type`*	Type of Edge TPU device to use when profiling each segment. Type may be "pcionly", "usbonly", or "any". Default is "any".
`--diff_threshold_ns` *`num`*	The target difference (in ns) between the slowest segment (upper bound) and the fastest segment (lower bound). When the partitioner finds a partition ratio where the difference between the slowest and fastest segment is lower than this, it stops. Default is 1000000 (1 ms).
`--partition_search_step` *`num`*	The number of operators that are added to or removed from a segment during each iteration of the partition search. Increasing this can speed up the search for a suitable partition, but also may overshoot a minimum. Default is 1.
`--delegate_search_step` *`num`*	Step size for the delegate search when compiling the model. Same as the `delegate_search_step` option with the compiler. Default is 1. (The `partition_with_profiling` tool always has `search_delegate` enabled.)
`--initial_lower_bound_ns` *`num`*	Initial lower bound of the segment latency. (Latency of the fastest segment.) By default, this is determined by the tool by segmenting the model with the compiler's heuristic-based approach and then recording the segment latencies.
`--initial_upper_bound_ns` *`num`*	Initial upper bound of the segment latency. (Latency of the slowest segment.) By default, this is determined by the tool by segmenting the model with the compiler's heuristic-based approach and then recording the segment latencies.

More detail about this tools is available in the **GitHub repo**.

To run an inference with your segmented model, read **Pipeline a model with multiple Edge TPUs**.

Compiler and runtime versions

A model compiled for the Edge TPU must be executed using a corresponding version of the Edge TPU runtime. If you try to run a recently compiled model on an older runtime, then you'll see an error such as this:

```
Failed precondition: Package requires runtime version (12), which is newer than this runtir
```

To solve this, update the Edge TPU runtime on your host system.

If you're unable to update the device runtime, you can instead re-compile your model to make it compatible with the older runtime version by including the `--min_runtime_version` flag when you run `edgetpu_compiler`. For example:

```
edgetpu_compiler --min_runtime_version 10 your_model.tflite
```

The following table shows the Edge TPU Compiler versions and the corresponding Edge TPU runtime version that's required by default. You can always use a newer compiler to create models compatible with older runtimes as described above.

Table 3. Compiler and Edge TPU runtime compatibilities

Compiler version	Runtime version required (by default)
16.0	14
15.0	13
14.1	13
2.1.302470888	13
2.0.291256449	13
2.0.267685300	12
1.0	10

> **Note:** Your compiler and runtime version determines which operations you may use.

You can check your compiler version like this:

```
edgetpu_compiler --version
```

You can check the runtime version on your device like this:

```
python3 -c "import pycoral.utils.edgetpu; print(pycoral.utils.edgetpu.get_runtime_version()
```

Help

If the log for your compiled model shows lots of operations that are "Mapped to the CPU," then carefully review the **model requirements** and try making changes to increase the number of operations that are mapped to the Edge TPU.

If the compiler completely fails to compile your model and it prints a generic error message, please **contact us to report the issue**. When you do, please share the TensorFlow Lite model that you're trying to compile so we can debug the issue (we don't need the fully trained model—one with randomly initialized parameters should be fine).